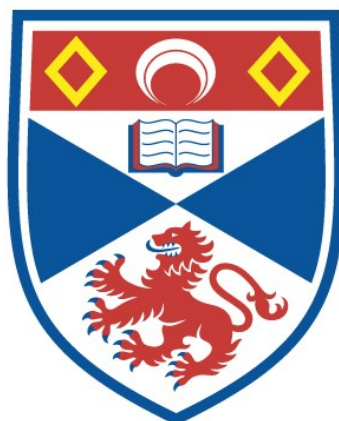


USING CONSTRAINTS TO IMPROVE GENERALISATION  
AND TRAINING OF FEEDFORWARD NEURAL NETWORKS:  
CONSTRAINT BASED DECOMPOSITION AND COMPLEX  
BACKPROPAGATION

Sorin Draghici

A Thesis Submitted for the Degree of PhD  
at the  
University of St Andrews



1995

Full metadata for this item is available in  
St Andrews Research Repository  
at:

<http://research-repository.st-andrews.ac.uk/>

Please use this identifier to cite or link to this item:

<http://hdl.handle.net/10023/13467>

This item is protected by original copyright

USING CONSTRAINTS TO IMPROVE  
GENERALISATION AND TRAINING OF  
FEEDFORWARD NEURAL NETWORKS:  
CONSTRAINT BASED DECOMPOSITION AND  
COMPLEX BACKPROPAGATION



A thesis submitted to  
The University of St Andrews  
in application for the Degree of  
Doctor of Philosophy

by

Sorin Draghici

May 1995



ProQuest Number: 10167231

All rights reserved

INFORMATION TO ALL USERS

The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if material had to be removed, a note will indicate the deletion.



ProQuest 10167231

Published by ProQuest LLC (2017). Copyright of the Dissertation is held by the Author.

All rights reserved.

This work is protected against unauthorized copying under Title 17, United States Code  
Microform Edition © ProQuest LLC.

ProQuest LLC.  
789 East Eisenhower Parkway  
P.O. Box 1346  
Ann Arbor, MI 48106 – 1346

Th  
B866



I, Sorin Draghici, hereby certify that this thesis, which is approximately 95,000 words in length, has been written by me, that it is the record of work carried out by me and that it has not been submitted in any previous application for a higher degree.

Date: 5.05.1995

Signature of candidate:

I was admitted as a research student under Ordinance No. 12 in 1991 and re-registered as a candidate for the degree of Doctor of Philosophy in 1992; the higher degree study for which this is a record was carried out in the University of St Andrews between 1991 and 1994.

Date: 5.05.1995

Signature of candidate:

In submitting this thesis to the University of St Andrews I understand that I am giving permission for it to be made available for use in accordance with the regulations of the University Library for the time being in force, subject to any copyright vested in the work not being affected thereby. I also understand that the title and abstract will be published, and that a copy of the work may be made and supplied to any bona fide library or research worker.

Date: 5.05.1995

Signature of candidate:

I hereby certify that the candidate has fulfilled the conditions of the Resolution and Regulations appropriate for the degree of Doctor of Philosophy in the University of St Andrews and that the candidate is qualified to submit the thesis in application for that degree.

Date: 11/9/95

Signature of supervisor:

## Acknowledgements

This thesis would not have been possible without the financial support of an ORS Award granted by CVCP and of a Research Studentship granted by University of St Andrews. I am very grateful for both. I am also grateful to my parents for teaching me the value of knowledge in the first instance and for the unbelievable sacrifices they have done to offer me the possibility to study. I would like to thank all my teachers and especially Dumitru Buturuga for the patience and talent with which he has introduced his students to a universe of Mathematics populated with polynomial dragons and exponential knights, Cristian Giumale who has opened a magic door into the wonderful world of Artificial Intelligence and Luca Dan Serbanati who has managed to convince us that an LALR grammar and a syntax directed compiler are not as bad as they might sound.

Many people helped me enormously during the three years I worked in St Andrews. Special thanks to Ron Morrison, Mike Atkinson, Ursula Martin, Mike Livesey for being with me in moments of great difficulty, Colin Allison, Dave Munro and Norman Paterson for putting up with all I did to the departmental computers and for helping me with all the technical problems I had during all these years. Many thanks go to Helen Bremner, Margaret Anderson and Brian McAndie for their help in solving the multitude of logistic problems I had to confront during my study in St Andrews.

Last but not least, I am grateful to my supervisor Mike Weir for accepting me as his PhD student. My work has been greatly influenced by his deep insight. Throughout the whole period he has managed to find many dead-ends thus helping me to remain on the right direction. Also, his meticulous insistence on apparently trivial details forced me to clarify my ideas. His remarkable command of the English language helped me to overcome the deficiencies of mine. Our relationship helped me enormously from a human point of view as well and I feel now that I am better prepared for confronting whatever difficulties I might face in the future. I believe that my patience, my endurance and my ability to cope with stress have been greatly improved during our collaboration.

## **Abstract**

Neural networks can be analysed from two points of view: training and generalisation.

The training is characterised by a trade-off between the 'goodness' of the training algorithm itself (speed, reliability, guaranteed convergence) and the 'goodness' of the architecture (the difficulty of the problems the network can potentially solve). Good training algorithms are available for simple architectures which cannot solve complicated problems. More complex architectures, which have been shown to be able to solve potentially any problem do not have in general simple and fast algorithms with guaranteed convergence and high reliability. A good training technique should be simple, fast and reliable, and yet also be applicable to produce a network able to solve complicated problems.

The thesis presents Constraint Based Decomposition (CBD) as a technique which satisfies the above requirements well. CBD is shown to build a network able to solve complicated problems in a simple, fast and reliable manner. Furthermore, the user is given a better control over the generalisation properties of the trained network with respect to the control offered by other techniques.

The generalisation issue is addressed, as well. An analysis of the meaning of the term "good generalisation" is presented and a framework for assessing generalisation is given: the generalisation can be assessed only with respect to a known or desired underlying function. The known properties of the underlying function can be embedded into the network thus ensuring a better generalisation for the given problem. This is the fundamental idea of the complex backpropagation network. This network can associate signals through associating some of their parameters using complex weights. It is shown that such a network can yield better generalisation results than a standard backpropagation network associating instantaneous values.

# Contents

## CHAPTER 1 Introduction

1.1 What are neural networks? .....	1
1.2 Why neural networks? .....	3
1.3 Brief history .....	4
1.4 Outline of the thesis .....	9

## CHAPTER 2 Training neural networks

2.1. Introduction .....	12
2.2. A trade-off: training speed vs. capabilities.....	12
2.2.1 Single layer networks.....	13
2.2.1.1 Threshold units .....	14
2.2.1.2 Linear units.....	17
2.2.1.3 Non-linear sigmoid units .....	20
2.2.2 Multilayer networks.....	21
2.3. Some training techniques .....	24
2.3.1 Variations of backpropagation.....	24
2.3.2 Other training techniques for feedforward networks.....	31
2.3.3 Network construction algorithms .....	33
2.4. Some factors influencing the training .....	37
2.4.1 Architectural issues .....	37
2.4.2 Dimensionality of the weight space.....	37
2.4.3 The pattern set.....	38
2.4.4 Conclusion. An ideal training problem.....	39
2.5. Assessing training speed.....	41
2.5.1 Assessing the speed of one trial .....	41
2.5.2 Assessing the average learning speed.....	45

## CHAPTER 3 Generalisation

3.1. Introduction. ....	48
-------------------------	----

3.1.1 A definition of generalisation for feedforward networks.....	48
3.1.2 Common methods for assessing generalisation. ....	51
3.2. Are there enough patterns in the training set? .....	51
3.2.1 An approximation point of view.....	51
3.2.2 A neural network point of view. ....	55
3.3. Good generalisation.....	56
3.3.1 Training without validation set. ....	56
3.3.2 The validation set.....	59
3.4. The use of the validation set.....	61
3.4.1 The resubstitution estimate.....	62
3.4.2 The test sample estimate.....	62
3.4.3 The V-fold-cross-validation estimate. ....	63
3.4.4 Conclusions regarding the use of the validation set.....	64
3.5. Validation of individual outputs.....	65
3.6. Generalisation as an approximation. ....	67
3.6.1 Generalisation with respect to classes of problems. ....	68
3.6.2 Generalisation with respect to types of underlying functions.....	69
3.7. Conclusions.....	71

## **CHAPTER 4 The constraint based decomposition approach**

4.1. Introduction .....	73
4.2. Time based decomposition (TBD) vs. constraint based decomposition (CBD).....	73
4.2.1 An example. ....	73
4.3. Theoretical framework .....	76
4.3.1 Definitions .....	76
4.3.2 Constraint implementation. CBD as a method to perform a dimensionality reduction in the weight space. ....	77
4.3.3 Search directed by subgoals .....	78
4.3.4 Search restricted by subgoals.....	79
4.4. Search directed by subgoals.....	79

4.5 Search restricted by subgoals. The Constraint Based Decomposition net. ....	80
4.5.1 The description of the CBD algorithm. ....	80
4.5.1.1. Two classes. ....	80
4.5.1.2. Proof of convergence for the CBD constructing algorithm (for binary outputs) ....	89
4.5.1.3. Classification of more than two classes. ....	91
4.6. Experiments. ....	97
4.6.1 Search directed by subgoals. Pattern presentation algorithm. ....	97
4.6.1.1. Constraint based decomposition versus standard training. ....	99
4.6.1.2. Investigating the search directed by subgoals. ....	102
4.6.2 Search restricted by subgoals. ....	105
4.6.2.1 Illustrating the algorithm with some toy-problems. ....	105
4.6.2.2. Constraint based decomposition versus standard backpropagation. ....	107
4.6.2.3. Constraint based decomposition versus divide and conquer network (DCN). ....	109
4.7. Conclusions and discussion. ....	110
4.7.1 Conclusions of the experiments. ....	110
4.7.2 The characteristics of the CBD architecture and training algorithm. ....	112
4.7.3 Relation to other work. ....	113

## **CHAPTER 5 Enhancements of the constraint based decomposition approach**

5.1 Introduction. ....	118
5.2 Improving the training speed. ....	118
5.3. Improving the training speed through locking detection. ....	119
5.3.1 Characterisation of the locking situations. ....	120
5.3.1.1 Two dimensions. ....	120
5.3.1.2 Three dimensions. ....	130
5.3.1.3 N dimensions. ....	131

5.3.2 Locking in CBD vs. locking in candidate elimination technique .....	139
5.3.2.1 Inconsistencies in the training set.....	141
5.4. Improving the training speed through avoiding problems beyond the possibilities of the current architecture.....	141
5.4.1 Linear separability.....	142
5.4.1.1 Two dimensions.....	143
5.4.1.2 N dimensions. ....	153
5.4.2 Conclusion.....	159
5.5. Efficient use of hyperplanes.....	159
5.5.1 The problem of redundant hyperplanes.....	159
5.5.2 Eliminating redundancy .....	162
5.6. Generalisation properties. The influence of the order of the patterns.....	167
5.6.1 General considerations .....	167
5.6.2 The influence of the order of the patterns .....	167
5.7. Other issues .....	173
 <b>CHAPTER 6 Experiments with the constraint based decomposition</b>	
6.1 Introduction .....	176
6.2 Hypothesis to be verified by experiments .....	176
6.3 Methods used .....	177
6.4 Locking detection.....	179
6.4.1 Experiments with locking detection.....	179
6.4.2 Conclusions of the locking detection experiments .....	186
6.5 Redundancy elimination .....	186
6.5.1 Experiments with redundancy elimination .....	186
6.5.2 Conclusions of the redundancy elimination experiments.....	191
6.6 The influence of the order of patterns.....	192
6.6.1 Study of the influence of the order of the patterns .....	192
6.6.2. Conclusions for the experiments investigating the influence of the order of the patterns. ....	194



6.7. General conclusions.....	195
<b>CHAPTER 7 Improving generalisation</b>	
7.1. Introduction .....	196
7.2. Motivation: automatic selection of the type of model we want.....	196
7.2.1. Parametric representation of signals .....	198
7.2.2. Signals determined by the training cycle .....	199
7.2.3. Arbitrary signal association .....	202
7.3. Approach .....	203
7.3.1. Signal space. A signal as a vector in signal space.....	203
7.3.2. Principal Component Analysis .....	203
7.3.3. Projections in signal space and their Fourier interpretation. Operators in signal space: rotate and scale.....	205
7.4. The complex network .....	206
7.4.1. The pre-processing filter. ....	207
7.4.2. The post-processing filter.....	208
7.4.3. The processing unit. ....	208
7.4.3.1. The approach.....	208
7.4.3.2. The designing of the processing unit.....	209
7.4.3.3. Detailed description of the processing performed by the net.....	213
7.5 Training the complex network .....	215
7.5.1. The complex backpropagation algorithm.....	215
7.6 Relation to other work .....	216
7.7. Conclusions .....	221
7.7.1 Main ideas in CBP: .....	221
7.7.2 Relation between the parallel approach, 'simple shaping' and 'rotation and scaling'.....	222
7.7.3 Caution regarding the usage of the complex backpropagation.....	222
<b>CHAPTER 8 Experiments with the complex network</b>	
8.1 Introduction .....	224

8.2 The objectives of the experiments .....	224
8.3. Short description of the experiments. Results in brief .....	226
8.3.1. Testing the training algorithm.....	226
8.3.2. Testing the training algorithm with a random I/O problem.....	228
8.3.3. Testing the generalisation .....	229
8.4 Calculating the error limit.....	230
8.5. Experimental details .....	234
8.5.1. Testing the generalisation .....	234
8.5.1.1 A single signal association.....	235
8.5.1.2 The association of two signals.....	251
8.6. Conclusions .....	256
8.6.1. Testing the training algorithm.....	256
8.6.2. Testing the training algorithm with a random I/O problem.....	256
8.6.3. Testing the general approach.....	256
 <b>CHAPTER 9 Conclusions and further work</b>	
9.1 General remarks .....	259
9.2. Limitations.....	261
9.2.1 Constraint Based Decomposition.....	261
9.2.2 The Complex Backpropagation.....	261
9.3 Further work .....	262
9.3.1 The Constraint Based Decomposition .....	262
9.3.2 Complex backpropagation .....	262
Bibliography .....	264
 <b>Appendix 1</b>	
Statistical analysis of experimental data.....	277
 <b>Appendix 2</b>	
Estimating the locking tolerance .....	286

### Appendix 3

Equations of complex backpropagation in radial/phase terms with two independent real activation functions.....	287
The derivation of the gradient descent equations in radial and phase terms .....	288

### Appendix 4

Details of some experiments with the complex backpropagation .....	298
1. Testing the training algorithm.....	298
General comments on the experimental set-up and results .....	298
Experiments.....	299
2. Testing the training algorithm with a random I/O problem.....	307

### Appendix 5

Software documentation.....	308
1. The simulation software for the Complex Backpropagation .....	308
1.1. Introduction .....	308
1.2. The source files .....	308
1.2.1 Where they are.....	308
1.2.2 The source file names and what they contain. ....	309
1.3 The make files.....	309
1.3.1 Where they are.....	309
1.3.2 Their name and their use .....	310
1.4. The simulation state files (SimState) .....	311
1.4.1 Their use and what they contain.....	311
1.4.2 Their structure. ....	312
1.5. Using the software .....	313
1.5.1 Introduction.....	313
1.5.2 Loading a network .....	313
1.5.3 Loading a simulation environment.....	314
1.5.4 Training .....	314
1.5.5 Saving the weight state .....	315

1.5.6 Loading a weight state.....	315
1.5.7 Testing.....	315
1.6 A Tutorial for Complex Backpropagation .....	316
1.6.1 Introduction.....	316
1.6.2 Using the simulator .....	316
1. Getting in the right place. ....	316
2. Making the executable. ....	316
3. Running the simulator .....	317
1.6.3 Running the program in the background .....	327
1.6.4 More about the existing files.....	329
2. Constraint Based Decomposition.....	331
2.1. Introduction .....	331
2.2. The source files .....	331
2.2.1 Where they are.....	331
2.2.2 The source files names and what they contain.....	331
2.2.3 Building different versions of the program. ....	332
2.3. Making the executable.....	333
2.4. The model file.....	333
2.5. Running the simulator.....	334
2.6 A Tutorial for using the CBD simulation software .....	334
2.6.1 Introduction.....	334
2.6.2 Using the simulator .....	335
2.6.3 More about existing files. ....	338
Appendix 6.....	339
Some solutions found by the Constraint Based Decomposition Algorithm .....	339

# CHAPTER 1

## Introduction

### 1.1 What are neural networks?

The present understanding of the human brain suggests that its basic elements are nervous cells called neurons. Block in [Block, 1962] states that: *"The doctrine [...] that the neurons are the functional units of the brain, is largely due to Ramon y Cajal and is now widely held among neurophysiologists"*.

A neuron has a cell body, a single long termination referred to as the axon and a large number of shorter endings referred to as dendrites. Usually, the axon is much longer than the dendrites and is also much longer than the cell body itself. The axon of a neuron comes into contact with dendrites of other neurons. The connection between two neurons is called the synapse and is the place of complex physiological phenomena which allow communication between neurons. The interaction between the pre-synaptic neuron and the post-synaptic one can be either of excitatory or inhibitory type. The efficiency of a synaptic connection can vary both between different synapses and in time for a given synapse and it is well accepted now that the adaptability of the synaptic efficiency plays a key role in the functioning of the brain.

The physiology of a generic neuron is very simple in principle although the phenomena involved are very complicated: when a neuron receives an excitation larger than a given threshold, it fires i.e. it produces an electrical impulse which is transmitted along its axon towards all the post-synaptic neurons the firing neuron is connected to.

The brain contains about  $10^{11}$  neurons of which about  $10^8$  i.e. approx. 1% are input/output neurons i.e. are connected to sensory or motor parts of the body [Block, 1962], [Hertz, 1991]. The nervous cells are rather slow in comparison with modern electronic devices. It has been suggested that the speed of a nervous impulse through the axon of a neuron varies between 5 m/s for small diameter neurons and 125 m/s for large neurons [Block, 1962]. According to more recent sources [Wender, 1994], the same speed varies between 0.5 m/s for small diameter unmyelinated neurons and 100 m/s for large myelinated ones. For comparison, the

electric signals in electronic circuits travel at about  $1.9786 \times 10^8$  m/s [Cheng, 1989]. Furthermore, the speed of biological neurons is affected by various other time delays such as the time a pulse needs to cross a synapse (about  $10^{-3}$  sec, [Block, 1962]). The firing rate of the neurons is further reduced by the time interval successive to a firing in which the neuron is not able to fire at all (the absolute refractory period of about 10 ms [Block, 1962]) or would fire only for an excitation much greater than the normal threshold (the relative refractory period of about 10-15 ms [Wender, 1994]). Thus, most of the neurons in the brain work at a firing frequency of about 100Hz. For comparison, the modern computers work at a clock frequency of tens or even hundreds of MHz.

In conclusion, the brain and the digital computer are very different. The brain has an extremely large number of very simple and very slow processing units. A computer has a single (or just a few) processing unit(s) able to perform quite complicated tasks and operating at extremely high speed. The computer is very good at formal symbolic manipulation (usually binary symbols) and not very much else. Any other task must be transformed into such a formal symbolic manipulation for the computer to be able to carry it out. Some tasks, such as numerical computation for instance, are easily translated into symbolic manipulation and the computer performs them very well. For other tasks though, especially real-world tasks like visual pattern recognition, speech processing or semantic reasoning, this translation is not simple. On the other hand, the brain is very poor at high precision symbolic numerical computation (with few exceptions) but is very good at the latter type of problems.

The efficiency with which the brain performs such tasks and the desire to build computers able to perform at the same level of performance stimulate interest in studying artificial devices with the same sort of properties as the brain's: high parallelism and high connectivity as opposed to sheer speed of sequential manipulation. Although at present the behaviour of a neural network is studied most often using computer simulations on Von Neumann computers, the ultimate goal of these investigations is to build highly parallel hardware able to perform well tasks which are currently out of the scope of the existing computers.

An artificial neural network can be defined as an information processing system formed with a large number of simple processing units which interact with one another through directed links with variable strength and which co-operate to solve a computational task. The units are sometimes called neurons. The variable strength

links were inspired by the biological hypothesis that the learning takes place by modification of the efficiency of the synapses (connections) between real neurons in the brain. Throughout this thesis, the artificial neural networks will be called neural networks or simply networks and the artificial neurons will be called neurons or units. This will not mean however that any biological or psychological plausibility is intended or implied. The approaches presented here are purely conceptual and their scope is intended to be the class of artificial neural networks not the real brain.

## **1.2 Why neural networks?**

Even though there are many differences between the real brain and neural networks, the study of neural networks can improve our understanding of the brain. Recent studies of neural network models (see [Stassinopoulos, 1994], [Alstrom, 1994]) offer a very promising approach to understanding the functioning of the real brain. Conversely, our continuously improving understanding of the real brain could suggest novel approaches to neural networks.

There has been in recent years, an increasing interest for parallel computers due to the decreasing costs of the hardware and the continuous improvement in technology. Because the parallelism is one of the fundamental features of a neural network, the study of such models can help build more efficient computers.

Recent years have brought real-world applications of connectionist models. Thus, neural networks have been shown to be able to compete with or even outperform various other techniques. Even if the performance of the neural network based systems is not better than that of the systems based on a classical approach to the field, the possibility of learning from examples characteristic to neural networks makes them a more convenient and very attractive alternative. Thus, this learning from example simplifies very much the initial set-up of a system and offers a much greater flexibility when external conditions change. Speech recognition and synthesis [Sejnowski 1986], hyphenation algorithms [Brunak, 1990], sonar target recognition [Gorman, 1988], navigation of a car [Pomerleau, 1989], image compression [Cotrell, 1987], visual pattern recognition [Fukushima, 1982], backgammon playing [Tesauro, 1990], signal prediction and forecasting [Lapedes, 1987] are only few of the practical applications in which neural networks were shown to outperform other techniques or to be a very interesting alternative at least.

### 1.3 Brief history

Since neural networks are so closely related to the study of the brain, their history is intertwined. Many reference papers are landmarks for both fields. Many important names in neural networks, especially in their early history, are the names of psychologists or physiologists.

In 1890, William James, a psychologist, was describing an "elementary" principle which is still extensively used in many contemporary neural network paradigms: *"When two brain processes are active together or in immediate succession, one of them, on reoccurring, tends to propagate its excitement into the other"* (see [James, 1890], p. 256). One can recognise here what we now know as Hebb's rule. Further on in the same section, James describes the functioning of the brain in the following terms: *"The amount of activity at any given point in the brain-cortex is the sum of the tendencies of all other points of discharge into it, such tendencies being proportionate (1) to the number of times the excitement of other points may have accompanied that of the point in question; (2) to the intensity of such excitements; and (3) to the absence of any rival point functionally disconnected with the first point, into which the discharges might be diverted"*. As Anderson and Rosenfeld point out in [Anderson, 1988], this is a model which uses hebbian synaptic modification and linear summation of synaptic inputs.

In 1943, McCulloch and Pitts published their paper describing the two state threshold neuron [McCulloch, 1943] and showed that any finite logical expression can be implemented by this type of neuron. As more recent physiological data show, the McCulloch-Pitts neuron is not a good model for the real neuron. However, this paper had a very important positive effect because it showed that complex computation can be implemented with simple binary elements. This stimulated computer scientists like John von Neumann (see [von Neumann, 1982]) and thus influenced the development of modern computers. At the same time, this paper presented a connectionist model in a formalism closer to the modern one.

In 1949, Hebb's book "The Organisation of Behaviour" marked another important step towards modern neural networks. In chapter 4 of this book, Hebb states that *"When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased"*. This is now known as Hebb's rule or hebbian learning.



In 1958, Rosenblatt presented the perceptron which was the first device able to learn to associate "specific responses to specific stimuli" [Rosenblatt, 1958]. Although the simplest perceptron presented in this paper has three layers of neurons (sensory units, association units and response units) and two layers of weights, the term perceptron is now used to designate a neural network with a single layer of adaptive weights. A possible reason for this is that Rosenblatt uses a winner-take-all design with a simple reinforcement learning rule and ultimately his perceptron can be functionally substituted by a single layer network.

In 1958, Selfridge described Pandemonium, a paradigm for learning in which a hill climbing maximisation technique is used to adapt the strength of the connection between units. His model uses a set of "cognitive demons" which are specialised dedicated devices that assess the presence of a given feature in the input. On a superior level, a "decision demon" chooses the output by selecting the cognitive demon with the highest output. Selfridge also suggests "rewiring" of old useless units or adding new units to improve the behaviour of the device.

Neural networks were brought a large step closer to what they are today in 1960 when B. Widrow and M. Hoff give the "Widrow-Hoff rule" (or LMS rule) in [Widrow, 1960]. This paper brought important new elements. A gradient descent was used in a supervised learning context to minimise an error function computed as the squared difference between the target and the actual output. This allows their device ADALINE (ADaptive LINEar Element) to continue training even after the correct classification is obtained and so to reach analogue targets.

In 1962, a detailed analysis of the Perceptron and a proof of a version of the perceptron convergence theorem was given in [Block, 1962]. By that time, a physical implementation of a perceptron had been built in the form of Mark I and encouraging experimental results had been obtained. In the same year, Rosenblatt proved the perceptron convergence theorem in his book "Principles of Neurodynamics" [Rosenblatt, 1962].

In 1969, "the correlograph", a model for a network able to do pattern association instead of the pattern classification of the perceptron is presented [Willshaw, 1969]. This model is very valuable for its simplicity and many modern VLSI implementations of pattern association devices are related to Willshaw's correlograph.

After initial spectacular results, neural networks failed to evolve at the same speed. The parallel development of the symbolic artificial intelligence field and its first successes gave an alternative topic for the general enthusiasm stimulated by the idea of a thinking machine. The development of neural networks came to an almost complete stop after 1969 when Minsky and Papert published their book *Perceptrons* ([Minsky, 1969]). Containing a very detailed analysis of the perceptron's limitations and being written in a very clear and concise style, the book was very well received and constituted a reason if not an excuse for diverting the efforts from neural networks to other more promising fields. Minsky and Papert's proof that perceptrons cannot solve linearly inseparable problems and their conjecture that "...our intuitive judgement [is] that the extension [to multilayer systems] is sterile" (p. 232) which came at the end of a very good book had a very long lasting influence on the development of the neural networks.

Although in the following years research in the field lost all of the enthusiasm associated with it in the early years, it didn't stop completely. In 1972, an associative memory device was independently proposed [Kohonen, 1972]; Anderson, 1972]. While Kohonen approached the problem from a mathematical point of view, Anderson did it from a physiological one to arrive at the same idea. As in [Willshaw, 1969], the case of associating patterns (as opposed to classifying them) is considered. The basic unit of this associative memory is a linear neuron. The memory can store a number of input patterns equal to the dimensionality of the space if they are orthogonal but much less if they are random. The idea of interference between patterns is put forward and the phenomenon is called "cross-talking".

The idea of modifying the weights by backpropagating the error started to appear in this period. According to [Hertz, 1991], the first to put forward this idea were Bryson and Ho [Bryson, 1969] in a control theory framework.

In 1973, the first approach to self-organisation was presented [von der Malsburg, 1973]. This paper was concerned with biological plausibility and tried to model the phenomena which take place in the visual cortex. Hebbian learning, normalisation of the weights and a more complicated activation function (the function which determines the response of a neuron to its inputs) which takes into consideration past activation values in a decay term are a few of the characteristics of von der Malsburg's network.

In 1974, the backpropagation idea was revisited in a PhD thesis, [Werbos, 1974] but the idea failed to catch general interest.

In 1975, a probabilistic model of the neuron inspired by biological data was introduced for the first time [Little, 1975]. Although a binary device, the neuron model used in this paper does not use the threshold in a deterministic way. It can fire even if its excitation is below the threshold or it can remain silent even if the excitation is above the threshold although this behaviour is less probable than the usual one. A system formed with such neurons can be described by a matrix of transition probabilities and it has been shown that there are long lasting (virtually stable) states in the evolution of such a system which can be used to store information. Two ways of modifying the duration of the state were proposed: modifying the internal threshold of the neurons and modifying the strength of the synaptic connection in a hebbian manner. The model introduced by Little and Shaw is very close to modern probabilistic networks such as Boltzmann machines and the long lasting states in the Little and Shaw model are close to the attractors in later Hopfield networks.

In a series of papers published between 1967 and 1980, S. Grossberg proposed a different approach to the general problem of learning in networks [Grossberg, 1967, 1968a, 1968b, 1969, 1972, 1976a, 1976b, 1980]. Models like "instar", "outstar", "avalanche", "embedding fields" and "ART" were proposed as a result of using a very mathematical approach and a biological inspiration.

In 1976, a model for "co-operative computation of stereo disparity" was proposed [Marr, 1976]. Although Marr later felt that this model is not appropriate ([Marr, 1982], sections 3.2 and 3.3), the model presented in this paper is interesting at least for its dynamic. After presenting the input pattern(s), the network is left to evolve (according to a connection matrix) until a stable state is reached. This type of dynamics is exactly the one used by later Hopfield networks.

In 1982, the stochastic binary Hopfield net was developed [Hopfield, 1982]. This first version of the Hopfield net used binary neurons (the output can be only 0 or 1) with connections specified by a connectivity matrix with zero diagonal (each neuron is connected to every other neuron but itself). When this matrix is symmetric (the connection between units  $i$  and  $j$  has the same strength as the connection between units  $j$  and  $i$ ), a potential (or energy) function can be defined. In this case, the process of updating the neurons is equivalent to decreasing this energy value. Thus, the network evolves towards stable states with low energy. The strong resemblance

between this behaviour and the behaviour of some well studied physical systems determined a new wave of interest in neural networks and brought into this field new categories of scientists like theoretical physicists and mathematicians. The Hopfield network was later enhanced with neurons able to produce a graded response which are closer to the real neurons ([Hopfield, 1984]).

In 1982, a self-organising system able to construct a topological feature map was introduced [Kohonen, 1982]. This model is remarkable both for its originality and for its similarity with the knowledge about the brain, where various topographic mappings have been identified (the visual cortex, the motor and somatosensory areas and the auditory cortex, for instance). The self-organising mechanism of the Kohonen network is very simple. Units are connected (with random weights) to their neighbours in an array of chosen dimensionality. Initially, when a pattern is presented to the network, the unit will respond randomly. The unit with the highest output is chosen (a winner-take-all policy) and the weights of all its neighbours are modified so their behaviour becomes closer to the winner's for the given pattern. This very simple strategy combined with some normalisation to prevent the weights from becoming very large is enough to form quite complicated topological maps.

In 1980, Fukushima et al. developed an idea originally introduced in 1975 (see [Fukushima, 1975]) and presented a device called "neocognitron" ([Fukushima, 1980]). The neocognitron approaches the visual pattern recognition problem from an engineering point of view but with a strong biological connection. The architecture and functioning of the neocognitron are inspired by the anatomy and physiology of the visual system. Although phenomena in the visual system seem to be more complicated now than it was believed then, the neocognitron architecture is extremely successful showing that the real nervous system can always be a valid source of inspiration for artificial systems. The neocognitron has been further developed since and shown to be able to recognise deformed and shifted hand-written characters [Fukushima, 1982].

In 1983, simulated annealing was proposed as a solution to the local minima problem in a non-neural context [Kirkpatrick, 1983]. In analogy with many physical phenomena, the simulated annealing idea was to use a parameter called temperature which influences the probability of transition from one state to another. When the temperature is high, transitions from low energy states to high energy ones are possible thus allowing the system to escape local minima in the search for the global minimum. Kirkpatrick presented the results of using this technique in

applications like designing the layout of the computer chips, routing of wires on printed circuit boards and the Travelling Salesman problem.

In 1985, simulated annealing was used by Ackley, Hinton and Sejnowski with a neural network model and the combination was called a Boltzmann machine [Ackley, 1985]. The Boltzmann machine is similar to a Hopfield network with the difference that the units are switched from one state to another in a stochastic manner with a probability given by a Boltzmann distribution. Once again, varying the temperature allows the system to free itself from local minima.

Also in 1985, training by backpropagating the error is rediscovered [Parker, 1985], [Le Cun, 1985]. Soon after, backpropagation gains notoriety due to Rumelhart and McClelland's "Parallel Distributed Processing" [Rumelhart, 1986]. The multilayer perceptron described in their book has become arguably the most used neural network paradigm since.

Late 80's is the period in which the first physical implementation and applications of neural network started to appear. An optical implementation of the Hopfield model [Farhat, 1985] and "a parallel network that learns to read aloud" [Sejnowski, 1986] are just two examples of attempts to use neural networks to solve real-world problems. The first VLSI implementations appeared at the same time. An analogue CMOS processing array able to do real-time visual computation was described in 1987 [Silviotti, 1987].

Since the late 80's, the neural network field has seen an explosion of interest materialised in thousands of papers. Fundamental results regarding the approximation capabilities of various types of networks, new training algorithms, physical implementations, commercial software simulations, dedicated hardware and very many other things have come and continue to come at a remarkable pace.

For a more detailed history since the beginnings see [Hertz, 1991], [Anderson, 1988] and [Rumelhart, 1986] and for a review of the most important results in the field since the late '80's see [Xu, 1992].

#### **1.4 Outline of the thesis**

This thesis explores two aspects of neural networks: generalisation and training.

Chapter 2 is a review of some training techniques. The presence of a trade-off between training properties and capabilities of the network is discussed. An aim to

combine the training speed and reliability of algorithms like the perceptron training algorithm with the capabilities of complicated architectures (usually not trainable by such algorithms) is put forward. Furthermore, some current approaches to assessing the training speed are reviewed and discussed and a more informative measure is proposed.

Chapter 3 reviews some issues regarding the generalisation and tries to give a definition for this well acclaimed but sometimes poorly understood property of neural networks. Issues such as good generalisation and generalisation as approximation are discussed. Furthermore, the relation between generalisation and the number of patterns in the training set and various validation techniques are investigated.

Chapter 4 presents constraint based decomposition (CBD), a new approach to training. In this approach, the training is defined as a constraint satisfaction problem. The constraint based decomposition approach is showed to improve the training of a standard weight changing algorithm (backpropagation). A new constructive algorithm using this approach is also presented. Some experiments designed to i) validate the ideas used in the approach and ii) compare the proposed algorithm with other well known algorithms are described. The relation between the proposed algorithm and other existing techniques is discussed.

Chapter 5 discusses some weaknesses of the constructive CBD algorithm and introduces some enhancements designed to eliminate them. The main enhancements are: i) improving the training speed through locking detection, ii) improving the training speed through elimination of non-separable problems and iii) improving the solution through elimination of redundant hyperplanes. Other characteristics of the CBD constructive algorithm are also discussed.

Chapter 6 presents experiments designed to illustrate and test the CBD approach. The efficiency of the enhancements is tested on various problems. The efficiency of the pattern presentation algorithm in controlling the generalisation properties of the solution is also tested.

In Chapter 7, a neural network using complex weights is designed starting from the motivation given by the generalisation ideas presented in chapter 3. The approach is to constrain the network to use some shape information given by the user-or requested by the problem. In this context, a priori constraints are used to improve generalisation.

Chapter 8 presents experiments designed to test the complex neural network presented in chapter 7. The abilities of the training algorithm to retrieve a given weight state and to train to random targets are investigated. A set of simple experiments designed to illustrate the approach is also presented.

Chapter 9 presents the conclusions of this thesis and recommended further work.

## CHAPTER 2

### On the training of static feedforward networks with supervised learning

#### 2.1. Introduction

This chapter will investigate some aspects of the training of a feedforward neural network. In §2.2, a trade-off between the training speed of the training algorithm and the capabilities of the architecture is discussed. In general, the more complicated the architecture, the more complex the training is. In §2.3, some training techniques are reviewed. Variation of backpropagation, other training techniques for feedforward networks and network construction algorithms are briefly discussed. In §2.4, some factors influencing the training are identified and an ideal training problem is defined using these factors. Section §2.5 is dedicated to a review of the most commonly used methods for assessing the training speed. Some drawbacks of these methods are discussed and a new method is proposed.

#### 2.2. A trade-off: training speed vs. capabilities

When discussing the capabilities of a neural network, there is an important distinction to be made between the architecture and the training algorithm. The capabilities of a given architecture of solving a given training problem reduce to the existence of a solution.

An architecture is able to solve a training problem if and only if a solution exists. The capabilities of a training algorithm reflect its possibilities to find the solution given that the solution exists.

Usually, the solution used is a single weight state found through the training algorithm such that the network gives the desired outputs when presented with the inputs of the patterns in the training set. Sometimes, a solution can be a set or a sequence of weight states as in [Chen, 1992] for instance.

The distinction between the existence of a solution and the search for a solution is the distinction between what a net can do and what a net can learn to do. What a net can do depends on the architecture (no of neurons, no of weights, the way the



neurons are connected, the activation function of individual neurons, etc.). What a net can learn to do depends on both the topology and the learning algorithm. As will be shown later in Chapter 4, it is also useful to define the learning algorithms as being composed of two elements: the weight change algorithm and the I/O presentation algorithm.

### 2.2.1 Single layer networks

The framework used in the following discussion is that of a fully connected feedforward network with one layer of active weights. The training is supervised i.e. the values of the targets are known and used during the training.

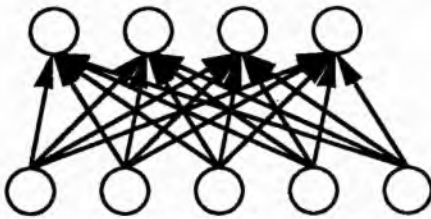


Fig. 1 A (single layer) perceptron.

A perceptron is presented in fig. 1. The computation performed by a perceptron is:

$$o_i = g(h_i) = g\left(\sum_k w_{ik} \xi_k\right) \quad (1)$$

where  $g$  is the activation function of a neuron,  $\xi_k$  are the inputs,  $o_i$  is an output and  $w_{ik}$  is the weight of the connection from unit  $k$  to unit  $i$ . A training instance is composed of an input vector  $\xi$  and an output vector  $o$ . The input vector will be referred to simply as a pattern.

The term perceptron is used by different people to designate different types of networks. In the following, this term will be used to designate a one layer feedforward network with a threshold activation function. The output is binary, for instance 0/1 or -1/+1.

The training problem is:

a) given a set of inputs and associated output values which are either +1/-1 or 0/1, is there a weight state so that the network using this weight state gives the correct outputs when the respective inputs are fed in?

b) If yes, can this weight state be found?

Note that the topology is known because there is just one layer, fully connected. The part a) of the problem above reflects the possibilities of the architecture and will be referred to as "possibilities". The part b) reflects the possibilities of the training algorithm alone (the solution is assumed to exist) and will be referred to as "training".

#### 2.2.1.1 Threshold units

In this case,  $g$ , the activation function of a neuron is a threshold function:

$$g(x) = \begin{cases} 0, & x < \text{threshold} \\ 1, & x \geq \text{threshold} \end{cases} \quad \text{or} \quad g(x) = \begin{cases} -1, & x < \text{threshold} \\ 1, & x \geq \text{threshold} \end{cases}$$

#### Possibilities

Without loss of generality, a network with a single output unit and  $-1/+1$  output values can be considered. One can divide the input patterns into two classes:  $C_+$  containing the patterns for which the output should be  $+1$  and  $C_-$  containing the patterns for which the output should be  $-1$ .

The equation:

$$\sum_k w_{ik} \xi_k = 0 \tag{2}$$

defines a hyperplane in the input space. According to equation (2), this hyperplane passes through the origin. This need not be always the case. To allow the hyperplane to be positioned anywhere in the input space, a bias term can be explicitly added or, alternatively, a supplementary input unit can be considered. In the latter case, the supplementary input unit is stuck at 1 and the form of equation (2) remains the same. This hyperplane will divide the input space into two half-spaces: a positive one and a negative one. All the inputs situated in one half-space will determine a positive value for the excitation of the neuron and all the inputs situated in the other half-space will determine a negative excitation. As a consequence, the perceptron will be able to solve only those problems for which there exists a hyperplane which separates the patterns with output  $+1$  from the patterns with output  $-1$ .

In other words, a problem is solvable by a simple perceptron with threshold units if and only if the problem is linearly separable. A very simple problem like the XOR is not linearly separable and therefore it cannot be solved by a simple perceptron. This limitation of the perceptron was shown by Minsky and Papert in [Minsky, 1969] and led to a loss of interest in neural networks for a long time.

There are two types of situations in which the problem is not linearly separable:

a) There exists a set of patterns containing patterns from both classes so that they are contained in a subspace with fewer dimensions of the input space. In this situation, the patterns are not in the general position i.e. this is a degenerate situation. An example for the 2D case is given in fig. 2.

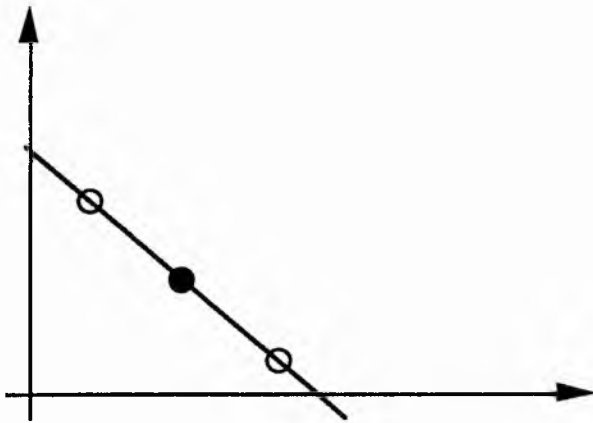


Fig. 2 The patterns are not linearly separable - the degenerate case. The axes are dimensions of the input space. The colour of the pattern shows the desired output (1 for black and 0 for white).

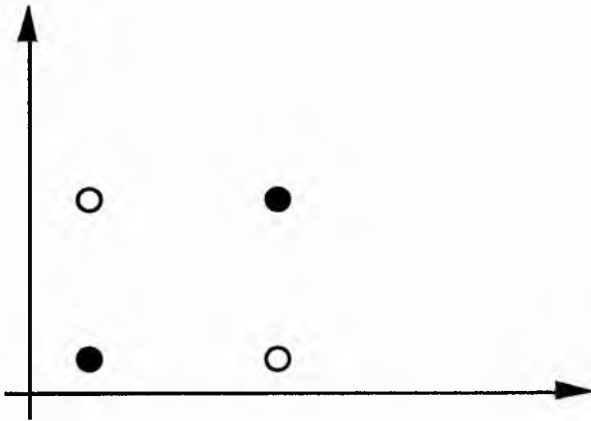


Fig. 3 The patterns are not linearly separable - the general case. The axes are dimensions of the input space. The colour of the pattern shows the desired output (1 for black and 0 for white).

b) There is no  $n-1$  dimensional subspace of the input space which contains the patterns and a separating hyperplane does not exist. An example for the 2D case is given in fig. 3.

### Training

The training problem reduces to whether a solution can be found in those situations in which a solution exists. The perceptron learning rule [Rosenblatt, 1962] changes the weight  $w_{ik}$  with the quantity  $\Delta w_{ik}$  given by:

$$\Delta w_{ik} = \eta (\zeta_i^\mu - o_i^\mu) \xi_k^\mu \quad (3)$$

where  $\eta$  is the learning rate,  $\zeta_i^\mu$  is the target of unit  $i$  for pattern  $\mu$ ,  $o_i^\mu$  is the actual output of unit  $i$  for pattern  $\mu$  and  $\xi_k^\mu$  is the  $k$ -th component of the input pattern  $\mu$ .

There is a theorem usually referred to as the perceptron convergence theorem [Block, 1962], [Minsky, 1969] which states that if the problem is linearly separable, the perceptron learning rule will find the solution in a finite number of steps.

In conclusion, the possibilities of the simple perceptron using a threshold function are very limited - only linearly separable problems - but the training has an important positive characteristic: if the solution exists, it will be found in a finite number of steps.

### 2.2.1.2 Linear units

In this case, the activation function  $g$  in (1) is the linear function  $g(x)=x$ , the outputs can take any values and are continuous functions of the inputs.

#### Possibilities

The output of unit  $i$  of a simple linear perceptron when pattern  $\mu$  is presented to the input units is given by:

$$o_i^\mu = \sum_k w_{ik} \xi_k^\mu \quad (4)$$

and this output should be equal to  $\xi_i^\mu$ .

If the patterns are linearly independent a solution can be calculated for the linear unit network using the pseudo-inverse method. According to this method, the weight between the units  $i$  and  $k$  is given by:

$$w_{ik} = \frac{1}{N} \sum_{\mu\nu} \xi_i^\mu (Q^{-1})_{\mu\nu} \xi_k^\nu \quad (5)$$

where

$$Q_{\mu\nu} = \frac{1}{N} \sum_k \xi_k^\mu \xi_k^\nu \quad (6)$$

Therefore, a sufficient condition for the problem to be solvable is that the patterns be linearly independent in input space.

A distinction can be made between the case in which the number of patterns  $p$  is less than or equal to the number of dimensions of the input space  $N$  and the case in which  $p$  is greater than  $N$ . If  $p \leq N$  and the patterns are linearly dependent, they only span a subspace of the input space. In this case, if a solution exists, the solution is not unique. Any weight of the form:

$$w'_{ik} = w_{ik} + a \xi_k^* \quad (7)$$

where  $w_{ik}$  is a solution,  $a$  is a real number and  $\xi_k^*$  is any vector perpendicular on the subspace spanned by the pattern set is a solution as well.

If  $p > N$  the patterns are linearly dependent because the maximum number of linearly independent vectors in an  $N$  dimensional space is  $N$ . In this case, a solution might or might not exist depending on the outputs required.

The output of a simple linear perceptron can be expressed as:

$$\begin{pmatrix} o_1 \\ o_2 \\ \dots \\ o_m \end{pmatrix} = \begin{pmatrix} \xi_{11} & \xi_{12} & \dots & \xi_{1n} \\ \xi_{21} & \xi_{22} & \dots & \xi_{2n} \\ \dots & \dots & \dots & \dots \\ \xi_{m1} & \xi_{m2} & \dots & \xi_{mn} \end{pmatrix} \begin{pmatrix} w_1 \\ w_2 \\ \dots \\ w_n \end{pmatrix} \quad (8)$$

where  $o$  is the output vector ( $m$  output units),  $\xi$  is the input vector ( $n$  input units) and  $w$  is the weight matrix. Each line of the matrix corresponds to an I/O pattern. Equation (8) is the matrix equation of a linear system which admits a solution if and only if the rank of matrix  $\Xi = (\xi)$  is equal to the rank of the augmented matrix  $\Xi' = (\xi)'$  and is less than or equal to the dimensionality of the input space  $n$ . The augmented matrix  $(\xi)'$  is obtained from  $(\xi)$  by adding to it the column vector  $o$ . For the same network with  $n$  input units and  $m$  output units, the augmented matrix is:

$$\Xi' = (\Xi | o) = \begin{pmatrix} \xi_{11} & \xi_{12} & \xi_{1n} & o_1 \\ \xi_{21} & \xi_{22} & \xi_{2n} & o_2 \\ \dots & \dots & \dots & \dots \\ \xi_{m1} & \xi_{m2} & \xi_{mn} & o_m \end{pmatrix} \quad (9)$$

If a solution exists and the dimensionality of the subspace spanned by the patterns in the training set is less than  $N$ , there are many solutions as in (7). If the dimensionality of the subspace is  $N$ , the solution is unique (if exists).

Most interesting problems do not satisfy the linear independence condition because usually the number of patterns is greater than the number of dimensions of the input space.

## Training

In the case of linear output units and analogue target values, one can define an error function or cost function by:

$$E(w) = \frac{1}{2} \sum_{i\mu} (\zeta_i^\mu - o_i^\mu)^2 = \frac{1}{2} \sum_{i\mu} \left( \zeta_i^\mu - \sum_k w_{ik} \xi_k^\mu \right)^2 \quad (10)$$

This error function depends only on the weights and the input patterns and is non-negative, and is zero when the targets are equal to the outputs. A gradient descent algorithm on the error-weight surface would require to change each weight with an amount proportional with the gradient of E at the given location. The weight change given by this algorithm is:

$$\Delta w_{ik} = -\eta \frac{\partial E}{\partial w_{ik}} = \eta \sum_{\mu} (\zeta_i^{\mu} - o_i^{\mu}) \xi_k^{\mu} \quad (11)$$

The change required by one pattern is:

$$\Delta w_{ik} = \eta (\zeta_i^{\mu} - o_i^{\mu}) \xi_k^{\mu} \quad (12)$$

Expression (12) is commonly referred to as the delta rule, Widrow-Hoff rule, the least mean square rule or the adaline (adaptive linear) rule [Rumelhart, 1986], [Widrow, 1960].

Although the expression (12) of the delta rule is identical with the expression (3) of the perceptron learning rule, their motivations, meanings and scopes are very different. The perceptron learning rule is valid for threshold units and it was derived from hebbian learning. According to the hebbian learning, the strength of the connection between two units is to be increased if the units fire at the same time. A natural extension of this rule is to change the strength of the connection between two units in proportion with the product of their activations [Rumelhart, 1986]. The hebbian learning does not impose any condition upon the activation functions of the neurons. On the other hand, the delta rule is valid only for units with continuous, differentiable activation functions and it was derived within a gradient descent framework. In order to perform a gradient descent, define an error function so that the error surface is non-negative and zero for the solutions and move on this surface, in the direction given by the gradient. Another difference between the perceptron rule and the delta rule is that if the solution exists, the perceptron learning rule is guaranteed to reach the solution in a finite number of steps whereas the delta rule converges towards the solution for an infinitely small learning rate.

It must be said that, even in this simple case, the convergence process can pose problems. If (and very often this is the case [Fernandes, 1994]) the smallest and the largest eigenvalues of the quadratic form (10) are very different (i.e. the error surface is very steep in some directions and very shallow in other ones) the

convergence exhibited by this technique can be "excruciatingly slow" [Hertz, 1991].

The conclusion for the simple perceptron with a linear activation function is that if the patterns are linearly independent in input space a solution exists and if a solution exists, the delta rule will converge towards it.

As an observation, any number of linear unit layers is equivalent to one because any linear combination of linear transformations can be expressed as a single linear transformation (see [Rumelhart, 1986] for instance).

### 2.2.1.3 Non-linear sigmoid units

In this case, the activation function  $g$  in (1) is a continuous, differentiable non-linear function such as the logistic function

$$g(x) = \frac{1}{1 + e^{-\beta x}} \quad (13)$$

or the hyperbolic tangent

$$g(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \quad (14)$$

### Possibilities

The sufficient condition for the existence of the solution is the same: if the patterns are linearly independent, a solution will exist. This is because the non-linear case can be seen as a linear case if the excitation values are used as targets.

### Training

The expression of the error (10) becomes:

$$E(\mathbf{w}) = \frac{1}{2} \sum_{i\mu} (\zeta_i^\mu - o_i^\mu)^2 = \frac{1}{2} \sum_{i\mu} \left( \zeta_i^\mu - g \left( \sum_k w_{ik} \xi_k^\mu \right) \right)^2 \quad (15)$$

and the expression of the weight change (11) becomes:

$$\Delta w_{ik} = \eta \sum_{\mu} (\zeta_i^\mu - o_i^\mu) g'(e x_i^\mu) \xi_k^\mu \quad (16)$$



where  $ex_i^\mu = \sum_k w_{ik} \xi_k^\mu$  is the excitation of unit  $i$  when pattern  $\mu$  is presented to the net. For the logistic function (13), the first derivative of the activation function  $g$  can be expressed as a function of  $g$  itself.

Although the sufficient condition for the existence of the solution is the same with the condition in the linear case (linear independence), the convergence properties of (16) are quite different. If the targets are outside the range of the activation function (for instance targets of 0 and 1 for a range of (0,1) or targets of -1 and 1 for a range of (-1,1)), the error surface can have local minima i.e. points in which the gradient is zero even if the error function is not [Hertz, 1991]. Furthermore, the convergence can be very slow because of the saturation of the activation function. That is because towards  $\pm\infty$  the activation function is very flat and its derivative is very small; therefore, the weight changes are very small [Burrascano, 1990b].

For the simple perceptron (one layer) the main difference between linear and non-linear activation functions is that the output is bounded in the non-linear case. The non-linearity of the activation function becomes essential in the case of multi-layer perceptrons. The non-linearity of the activation function is the element which allows multi-layer networks to solve problems which are not linearly separable and this is why the discovery of a learning rule for the multilayer net determined a new wave of interest in neural networks.

### 2.2.2 Multilayer networks

#### Possibilities

A multilayer perceptron with a linear activation function is equivalent to a single layer perceptron. Therefore, the possibilities of a multilayer net will be the same as the possibilities of the single layer perceptron. Thus, only the non-linear case remains interesting and needs discussion.

Let us consider a network with two layers and one output unit. A layer refers to a layer of trainable weights. Thus, a two layer network will have three layers of neurons but only two layers of active (trainable) weights because the first layer of neurons is used just to store the input values. For such a network, the output can be expressed as:

$$o = g\left(\sum_j w_{ij} g\left(\sum_k w_{jk} \xi_k^\mu\right)\right)$$

where  $k$  refers to units in the first layer and  $j$  to units in the second layer. For an arbitrary weight state, this expression is a function of the input. As a consequence, a feedforward network will be able to implement only functional relationships.

If no one-->many associations are present in the given I/O pattern set, the set can be seen as a set of samples of a desired I/O function. In this approach, the network's I/O function should approximate the desired I/O function. The possibilities of the multilayer perceptron have been studied from this point of view.

Justifications for the possibilities of a multilayer nets are given by Kolmogorov's superposition theorem (1957) and its refinements. These results assume different unknown activation functions for each function to be approximated. They also specify an exact upper limit for the number of intermediate elements (implemented by hidden units in the case of a neural network). However, in the neural network framework, the usual situation is that there is a single fixed activation function and the number of hidden units is not limited a priori.

Hornik in [Hornik, 1989] demonstrates a result which is more appropriate in this framework: a standard multilayer feedforward network with a single hidden layer using arbitrary squashing functions is capable of approximating any Borel measurable function from one finite-dimensional space to another to any desired degree of accuracy, provided sufficiently many hidden units are available.

This result must be seen as an answer to the question regarding the existence of the solution. Provided that enough hidden units are available, in general, a solution weight state will exist. However, the theorem does not give a method to estimate the necessary number of hidden units given a function to be approximated nor does it give a training algorithm which finds the solution starting from arbitrary weights.

Very recently, Hornik in [Hornik, 1993] showed that the approximation capabilities of a standard feedforward net with a single hidden layer are better in various further ways than previously shown.

Following the original idea suggested by Kolmogorov's theorem, Sprecher in [Sprecher, 1993] gives a method for constructing the activation functions of a two layer network able to approximate any given I/O mapping. The main strengths of this result are that i) the activation functions of the first layer are fixed and do not depend on the dimensionality of the input space, ii) the number of neurons in the each layer is known ( $2n+1$  in both hidden and output layers when  $n$  is the

dimensionality of the input space) and iii) only the activation functions of the output layer depend on the function to be approximated.

As a conclusion, the possibilities of a multilayer perceptron with non-linear activation functions are much larger than those of a single layer perceptron. Relatively simple architectures (at most three layers) can approximate to any degree of accuracy various classes of functions.

In particular, for any finite discrete set of I/O patterns there exists an architecture and a weight state able to implement it. This is because, for any finite discrete set of I/O patterns one can define a continuous function on a compact interval of the input space and this function can be approximated to any degree of accuracy as shown in [Hornik, 1993].

### Training

In the case of a network with one hidden layer, the expression of the error (15) becomes:

$$E(\mathbf{w}) = \frac{1}{2} \sum_{i\mu} (\zeta_i^\mu - o_i^\mu)^2 = \frac{1}{2} \sum_{i\mu} \left( \zeta_i^\mu - g \left( \sum_j w_{ij} g \left( \sum_k w_{jk} \xi_k^\mu \right) \right) \right)^2 \quad (17)$$

It is clear that the error function is a differentiable function of the weights (as a composition, sum and product of differentiable functions). The error is always positive or zero as a sum of squares and is zero when the outputs are equal to the targets. This function defines an error surface on which one can perform a gradient descent.

For output unit, the expression of the weight change (16) becomes:

$$\Delta w_{ij} = \eta \sum_{\mu} \delta_i^\mu out_k^\mu \quad (18)$$

$$\text{where } \delta_i^\mu = (\zeta_i^\mu - o_i^\mu) g'(ex_i^\mu) \quad (19)$$

$ex_i$  is the excitation of unit  $i$  and  $out_k^\mu$  is the output of the hidden unit  $k$ .

For hidden units, the weight change is:

$$\Delta w_{jk} = \eta \sum_{\mu} \delta_j^\mu \xi_k^\mu \quad (20)$$

$$\text{where } \delta_j^\mu = g'(ex_i^\mu) \sum_i w_{ij} \delta_i^\mu. \quad (21)$$

It can be seen in (21) that the delta values are propagated backwards through the network using the same weights. The result can be easily extended to any number of hidden layers. This training algorithm is called backpropagation and, according to [Hertz, 1991] was invented independently several times, by Bryson and Ho (1969), Werbos (1974), Parker (1985) and Rumelhart et al. (1986).

The standard backpropagation training without momentum [Rumelhart, 1986] has numerous problems: it is slow, it can be trapped in local minima, the architecture must be known in advance and there is no possibility to distinguish between a failure caused by inadequate training parameters and a failure caused by an inadequate architecture. These problems, some methods to overcome them, and other training techniques will be reviewed in the next section.

The conclusion of this section is that for these algorithms, all based on a fixed architecture, there is a trade-off between the architectural possibilities of a network and the training algorithms which can be performed on that particular architecture. The simple perceptron with a threshold activation function is able to solve only a very limited class of problems but its training has an important positive feature: it is guaranteed to find a solution in a finite number of steps. As the possibilities of the architecture increase by adding more layers, this positive characteristic is lost. The multilayer perceptron with non-linear activation can approximate almost anything in theory but its training can fail even on simple problems like XOR. The aim of an ideal training technique could be to combine the training characteristics of a simple architecture with the possibilities of a more complicated one.

## **2.3. Some training techniques**

### **2.3.1 Variations of backpropagation**

There are various methods designed to improve the training characteristics of the backpropagation. Such methods are reviewed by Xu, Kłasa and Yuille in [Xu, 1992] and Hertz, Krogh and Palmer in (Hertz, 1991). Some of these methods are presented in the following.

## **Other error functions**

The error function (15) can be substituted by any other differentiable function of the weights which is minimised when the targets are equal to the outputs. A corresponding weight change rule can then be derived.

The error function (15) can determine difficulties for the training. It has been shown, (see [Brady, 1988]) that gradient descent with an LMS error function can fail to separate on problems in which the perceptron learning rule would succeed. In such situations, a weight vector which separates does exist but it is different from the weight vector which minimises the error function (17). The problem is solved if a threshold least mean square (LMS) error function is used. The difference between the threshold LMS function and the standard error function is that the threshold LMS function is zero for values towards the extremes, beyond the target values. It has been shown that backpropagation with such an error function is guaranteed to find a separating weight vector, in finite time and starting from any initial weight configuration [Sontag, 1989].

In [Burrascano, 1990a], the error function is defined using a Chebyshev norm instead of the  $L_2$  norm. This is equivalent to backpropagating only the error signals determined by the largest component of the error vector determined by the output units. Although backpropagation with this error function converges, it is not clear that the convergence properties are better than those of the standard backpropagation.

## **On-line updating.**

A strict implementation of the gradient descent requires the backpropagation weight changing mechanism to update the weights in a batch mode. According to this batch updating rule, the weight changes corresponding to different patterns in the training set are accumulated and the weights are updated only once during a training epoch. In this way, the same weights are used to calculate the error values for all patterns.

If the weight update is made after each pattern (on-line updating) instead of being made after the entire training set (batch updating) some beneficial effects can appear. Thus, choosing each pattern randomly from the training set will produce small random fluctuations which can let the system get out of local minima, if these minima are not too deep [Xu, 1992; Rumelhart, 1986].

**Adding artificial noise.** There are at least three different places where the noise may be added. The noise can be added i) in the weight updating phase ii) in the weights or iii) in the training patterns.

In i) the weights are updated with the values given by the chosen gradient descent technique plus some noise. In ii) the weights are substituted by their values plus some noise at all times i.e. for both forward and backward steps. This is a good model for hardware implemented neural networks where the noise is unavoidable. This type of noise has been reported to have good effects upon both the training and the generalisation [Murray, 1992]. In iii) the patterns are substituted in each epoch by their values plus a random noise.

In every case the noise can introduce fluctuations which let the system get out of local minima. The improvement in generalisation is explained by the fact that the network will use the fundamental features of the patterns which are the same independently of the noise, instead of using some superficial characteristics which will be masked by the noise. From the point of view presented in the chapter on generalisation, this improvement is due to the fact that the actual training is performed on very many different data points created by adding noise to the original, pure, data points. This is equivalent to training the network with many functions whose shapes differ very little from the shape of the target function. This ensures that the number of possible model functions which pass through this many data points is reduced, which means that the probability of obtaining a function close to the target function is increased.

**Annealing.** Simulated annealing techniques are global optimisation techniques which use stochasticity to avoid local minima. The algorithms in this category use a parameter called temperature which characterise the state transition probability. The temperature is initialised with a large value and decreased as the training evolves. A particular implementation of simulated annealing is the Boltzmann machine. In this case, the gradient is calculated as in backpropagation's case but the weights are updated using a probabilistic rule. This rule will change the weights according to the gradient only with a given probability which in turn depends on the temperature. The good training characteristics of the Boltzmann machine are obtained by allowing weight changes which are uphill the direction given by the gradient in the beginning, when the temperature is high. As training continues, the temperature is lowered and the probability of uphill weight changes is diminished. [Atkin, 1989], [Ackley, 1985].

**Momentum.** The shape of the error-weight surface on which the gradient descent is performed can be very complicated and/or very unfavourable at times. A classical example is a long and narrow valley. The sides are steep giving a large value for the transversal component of the gradient and the valley floor has a very shallow slope giving a very small longitudinal component (the error function has very different eigenvalues). The system will jump from one side of the valley to the other with a very small progress along the direction of the valley.

A simple solution to this problem is the adding of a momentum term in the expression used to calculate the weight update. The momentum term has beneficial effects upon the overall behaviour of the system. Thus, the momentum makes the weight change increase if the direction remains constant, increasing in this way the convergence speed. Momentum also damps oscillations and smoothes the trajectory ([Plaut, 1986]).

### **Different gradient techniques**

The problem of a complicated error surface can be more successfully confronted by different gradient techniques such as steepest descent, conjugate gradient and Newton's method [Press, 1992].

The steepest descent uses a line minimisation along the direction given by the local gradient. Then, the local gradient is re-evaluated and the process is repeated. It can be shown that any current search direction (local gradient) is perpendicular on the previous one (otherwise the gradient would have a component along the old direction and the current point would not be a minimum). Because of this fact, the method takes steps in mutually perpendicular directions which can be inefficient. In fact, steepest descent is reported to be considerably faster than ordinary back-propagation ([Watrous, 1987] and [Kramer, 1989] cited by [Hertz, 1991]).

In the case of conjugate gradient descent, the idea is to minimise along non-interfering directions. Two directions are said to be non-interfering if once a minimum along one direction has been found, it will not be affected by the minimisation along the other direction. In other words, a minimisation along a direction will not change the component of the gradient along the other direction. In this way, a minimum can be found along one direction at a time. In the previous case of a long and narrow valley in 2 dimensions, conjugate gradient descent starts with the direction given by the steepest component of the gradient and finds the minimum of a cross-section of the valley (which will set the system on the bottom

of the valley) and then finds the minimum along the long axis. The most recent algorithm in this category is scaled conjugate gradient which eliminates the learning rate and the momentum as user dependent parameters [Moller, 1993].

A more complicated variation of this technique is the Newton's technique which uses the inverse of the Hessian in order to calculate the weight change. The error function can be expanded about the current point  $x_0$ .

$$E(x) = E_0 + (x - x_0) \cdot \nabla E(x_0) + \frac{1}{2}(x - x_0) \cdot H \cdot (x - x_0) + \dots \quad (22)$$

where  $H$  is the second derivative Hessian matrix:

$$H_{ij} = \frac{\partial^2 E}{\partial x_i \partial x_j} \quad (23)$$

evaluated in  $x_0$ . Differentiating (22) gives:

$$\nabla E(x) = \nabla E(x_0) + H \cdot (x - x_0) + \dots \quad (24)$$

In the minima of the error-weight surface  $E(x)$ , the gradient  $\nabla E(x)$  is zero. If one ignores the high-order terms in (24), one obtains:

$$\begin{aligned} \nabla E(x_0) + H \cdot (x - x_0) &= 0 \\ x &= x_0 - H^{-1} \nabla E(x_0) \end{aligned} \quad (25)$$

as an estimation of the position of the minimum. Newton's method uses the second equation in (25) iteratively to find the minimum of the error-weight surface.

The method allows an improvement of the training speed by an order of magnitude but is heavy from a computational point of view. An alternative is to approximate Newton's technique by using only an approximation of the Hessian. Depending on the approximation used the method is called the quasi-Newton or pseudo-Newton rule ([Parker, 1987]).

In Fahlman's quickprop algorithm, the weight changes are calculated using two assumptions: i) the section through the error surface in the direction of a weight (i.e. the error-weight curve for each weight) is a concave parabola and ii) that the change in the slope of the error curve as seen by each weight is independent of the other weights which are changing at the same time. It has been shown that



quickprop is much faster than backpropagation and scales up much better on some problems [Fahlman, 1988].

**Variable learning rate.** A single uniform and constant learning rate is unable to suit a complex error surface. If the learning rate is too small locally, the learning will be very slow while if the learning rate is too large locally, the learning will overshoot the target which again slows down the learning or even makes it diverge. A simple and very common policy is to start with a large learning rate (which could overshoot the target but could also allow escaping from local minima) and to decrease it as the training proceeds.

Weir in [Weir, 1991] proposes a method for self-determination of adaptive learning rates which uses the height and gradient information to ensure that the target is never overshoot.

Various other techniques have been studied ([Jacobs, 1989], [Fahlman, 1988], [Samad, 1990], [Vogl, 1988]).

**Avoiding saturation.** Due to the non-linear sigmoid function usually used as the activation function of a neuron, the weight update given by the gradient descent rule can be very small even if the error is still relatively large. This happens if the neuron is excited with large positive or negative quantity. On the sigmoid's graph presented in fig. 4, this regime corresponds to the horizontal parts going towards 1 and -1 (or 0 and 1) as the excitation goes towards infinity or minus infinity. In these regions, a large change in excitation will determine only a small change in the output i.e. the first derivative is almost zero. In analogy with the behaviour of some electronic devices this effect is often called saturation. The basic remedy is to let the parameter  $\beta$  which determines the shape of the sigmoid (see expression (13)) be adjustable so that when the learning falls into a premature saturation the shape of the sigmoid is adjusted to allow the error to back propagate efficiently [Yamada, 1990].

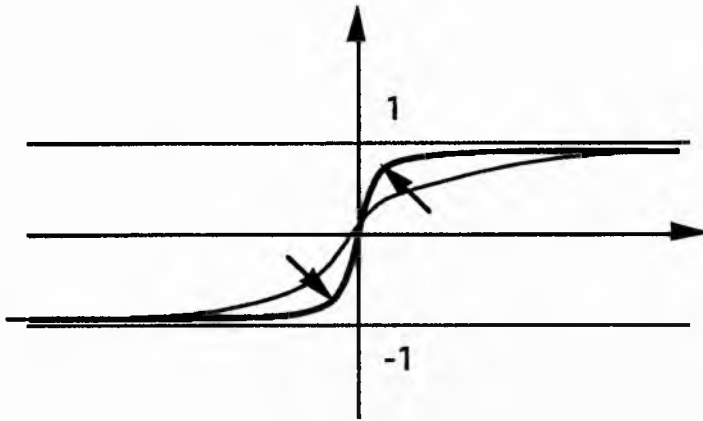


Fig. 4 A sigmoidal function taking values between -1 and 1. The arrows show the change in the shape determined by a larger  $\beta$

A different approach is to calculate the weight changes according to a different formula which does not saturate. This method is presented in [Burrascano, 1990b] and is equivalent to training with a more complicated activation function. As the behaviour of this function is engineered to be the same with the behaviour of the standard sigmoid during the post-training phase and the actual values of the outputs are not needed during the learning phase the value of this more complicated activation function is never actually computed. Its purpose is just to eliminate the saturation in the learning phase.

### **Weight adjustment.**

Inappropriate initial weights can place the learning near a local minima or determine a premature saturation. Both cases could lead to a very long training time or even failure to converge. The appropriate selection of the initial weight values is not an easy task. One idea is to choose the initial weights in such a way that all hidden units are 'scattered' uniformly in the input pattern space [Nguyen, 1990]. Each hidden unit is seen as performing a linear approximation near a point in input space. Having the hidden units scattered means that the points around which they perform this linear approximation are not closely grouped together but distributed all over the interest area in input space. Thus, the initial weights are chosen so that the units exploit their range of linear activation from the beginning.

A different idea is to incorporate in the initial weight state a priori information about the task to be learned. This offers the advantage of starting from a position in the weight space relatively close to the solution but asks the user to have some a priori

knowledge about the task. A slightly different approach uses prototypes i.e. representative samples from various classes, to initialise the weights. In this case, one not only knows some properties of the patterns to be learned as in the previous case, but one also knows a prototype of each input class ([Joerding, 1991], [Denoeux, 1993]).

### **2.3.2 Other training techniques for feedforward networks**

Although backpropagation is still the most widely used algorithm for training multilayer perceptrons, there are other methods which are not based on backpropagating the error. Some of them will be reviewed in the following.

#### **Derivative estimation by perturbation**

The derivative estimation by perturbation method first injects some perturbations in the network, propagates them forward and calculates the change in error caused by them. Then, this change in error is used to approximate the gradient of the error function. The idea has two variations, to inject the perturbation locally, at the level of one neuron as in MRIII [Andes, 1990], or globally as in model-free distributed learning [Dembo, 1990].

#### **Direct update of the weights**

The method of directly updating the weights by perturbation eliminates the need for gradient computation. The weights are simply changed by an arbitrary quantity and the new error value is calculated. If the change has determined a decrease of the error, the change is accepted. If error has been increased by the weight change, the weight change is discarded and some other change will be tried. Again, there are two variations of this idea, one which applies the weight changes on an individual basis (to each weight) and one which considers a perturbation matrix which perturbs the weight matrix [Baba, 1989].

#### **Genetic algorithms**

Another approach to training is genetic algorithms. In this method, the weight state and/or the architecture are encoded into a binary string called a chromosome. For each chromosome, a fitness measure is calculated. This fitness measure is inversely proportional to the error on the training set. Initially, a population of chromosomes is generated randomly and subsequently new generations are created by using a set of genetic operators. There are two categories of genetic operators: cross-over

operators which combine bits from two chromosomes to create a third one and mutation operators which randomly change some bits in the chromosome it acts upon. At each generation, some individuals will die if their fitness value is below a certain threshold. After a certain number of generations, a solution is obtained by choosing the best fit (or any) member of the current generation. The main disadvantages of the genetic approach are the extreme sensitivity to the binary codification of genes, the genetic operations and the rate at which they are applied and the calculation of the fitness values. Very small changes in any of these elements could lead to a very slow evolution speed or even failure. Genetic algorithms are described in [Chang, 1990], [Dodd, 1990], [de Garis, 1990] and others.

### **Basis functions**

In this case, the network approximates the desired function using a set of functions which form a basis in the function space:

$$F(W, X) = \sum_{i=1}^m W_i \Phi_i(X)$$

where  $\Phi_i$  are the basis functions,  $W_i$  are weights and  $X$  is the input pattern. Each neuron on the hidden layer is fully connected to each input neuron and implements a basis function. The output layer performs a linear summation of the basis functions supplied by the hidden layer. The main difference from the backpropagation variations is that the weights from the input layer to the hidden layer are not trained by error propagation but they are calculated directly from the training samples. The weights to the output layer, however, can be trained using the delta rule or calculated directly by solving a linear system.

The basis functions can be localised or not. If the functions are local, their effect is present only in a limited region of the input space. The most commonly used type in this category is the radial basis function type in which the basis functions are Gaussian functions [Broomhead, 1988], [Musavi, 1992], [Moody, 1989], [Poggio, 1990], [Girosi, 1990]. Different non-localised basis functions can be used, as well. A possible choice for the basis functions is a set of orthogonal polynomials.

## **Converting decision trees**

A classification network performs the same task as a classification tree. Several authors have shown that it is possible to construct a neural network equivalent to a given classification tree. This network can be used as it is or as a starting point for a backpropagation training or other training techniques. The approach presents the advantage that many classical techniques for designing tree classifiers can be directly used in building neural networks. A disadvantage could be that usually, the construction of the classification tree is a process with heavy computational demands. The connection between tree classifiers and neural networks is explored in [Sethi, 1990a], [Sethi, 1990b] and [Sirat, 1990].

## **Learning from examples and queries**

The techniques presented until now, are based on a training set which is independent of the learning process. Query learning uses a partial training with very few examples after which the algorithm calculates some 'interesting' points in the input space and the output values for those inputs are asked. The advantages of this approach are firstly that a good approximation can be constructed using a small training set and secondly that the algorithm itself can find the points which are more important (such as the boundary points in a classification problem) and can pay more attention to them. The disadvantage is that the approach requires the existence of an oracle i.e. a mechanism able to give the correct output for any input point the algorithm might ask. There are many real world problems for which such an oracle does not exist. Two examples of query learning algorithms are [Hwang, 1990] and [Baum, 1991].

### **2.3.3 Network construction algorithms**

To overcome the difficulty of estimating the architecture necessary for solving a given problem, a number of network construction algorithms have been proposed.

The upstart algorithm proposed by Frean in [Frean, 1990] uses linear threshold units. The method adds units as the training proceeds. New units are introduced between the input layer and the output unit. The role of these units (the daughter units) is to correct mistakes made by the output unit (the parent unit) and this is done by using large enough weights which override the wrong excitation of the output unit. Subsequently, the errors of the daughter units are corrected by other

units and the process continues until all the patterns are correctly classified. The resulting multilayer architecture can be converted into a two layer architecture.

The weight update is performed for one unit at a time and consequently, a perceptron learning can be used. Actually, the upstart algorithm uses an improvement of the perceptron learning called the pocket algorithm. The perceptron will find a solution if the classes are linearly separable but will offer a very poor weight state if the classes are not linearly separable. The pocket algorithm keeps a copy of the weight state which has lasted longest without being changed. This weight state will give the minimum possible number of errors with a probability approaching unity as the training time increases. However, there is no bound known for the training time actually required.

The tiling algorithm proposed by Mezard and Nadal in [Mezard, 1989] (from [Hertz, 1991]) is based on the idea that if two patterns have different targets at the output layer, then their final internal representations on each hidden layer must be different. The algorithm starts each layer with a master unit that does as well as possible on the target task. If not all patterns are correctly classified, then there is a subset of input patterns which produce the same output and contains patterns from different classes. An ancillary unit is added on the same layer and trained on this subset so that as many of them as possible are separated. Ancillary units are added on a layer until the internal representation of that layer is faithful. The algorithm terminates when the master unit of a layer classifies correctly all patterns in the training set.

Any algorithm which can separate two classes can be extended to perform n-class separation by performing n separations between each class and an 'other' class which includes all the other n-1 classes. For this reason, many algorithms are presented only in the two-class version. An algorithm which deals directly with multiclass classification is presented in [Knerr, 1991]. According to this algorithm, the building and training phase proceeds in three steps:

- a linear separation of each class from all others is attempted: this results in successful neurons, which are kept, and in unsuccessful neurons, which are discarded;
- a pairwise linear separation of the classes which were not separated during the previous step is attempted; again, the resulting successful neurons are kept, whereas the unsuccessful ones are discarded;

- the classes which are still not separated (because they are non-linearly separable) are separated pairwise by piecewise linear decision surfaces. This is performed by building a binary decision tree through successive splits of the input space until all examples are correctly classified. The tree can be subsequently pruned to optimise the trade-off between performance and complexity.

Marchand in [Marchand, 1990] shows that a particular sequential learning which eliminates patterns from the training set as the training proceeds, gives always a linearly separable internal representation of the pattern set. The algorithm starts with just one hidden unit and a weight state which classifies correctly all the patterns from one class and some patterns from the opposite class and tries to modify the weights so that the number of classified patterns is maximised. Then, the algorithm eliminates the patterns which have been correctly classified, adds a new hidden unit and reiterates starting from a misclassified pattern. The failure of a training is detected theoretically because the perceptron will eventually cycle through the weight states.

The cascade correlation algorithm proposed by Falhman and Lebiere in [Fahlman, 1990] builds a cascade network in which each neuron is connected to all inputs and all previous hidden units. The input-side weights of the most recently added hidden unit are adjusted so that the correlation between its output and the residual error at the output level be maximum. This is to ensure that each hidden unit is as useful as possible. Subsequently, these weights will be frozen so that the unit becomes a permanent feature detector in the network. The training of the output weight is done using the quickprop algorithm.

A related constructive algorithm is described in [Brent, 1991]. The algorithm builds a decision tree. The hyperplane which determines each new split is determined so that the information associated with it is maximised. This is a heuristic method to ensure the size of the decision tree implemented by the net is minimal. The maximisation of the information gain (or the minimisation of the entropy) is performed with a gradient descent algorithm.

An algorithm which starts with a single hidden unit and attempts to train all the patterns in the training set is proposed in [Hirose, 1991]. The training failure is detected by checking the total error every  $N$  epochs. The training is halted if the error has not decreased by at least  $p\%$  in the last  $N$  epochs. In this situation, a new hidden unit is added. The network is said to have converged when the total error is less than a pre-determined error limit. Due to the fact that this algorithm can lead to

growing very large nets, a second phase is used. In this second phase, the removal of hidden units (in the reverse order of adding) is attempted. The most recently added hidden unit is removed and the net is re-trained. If the training is successful, another hidden unit is removed and the net re-trained. If a training fails, the last architecture able to solve the problem is chosen as the final solution of the problem.

The extention proposed by Baffes and Zelle [Baffes, 1992] grows multilayer networks capable of distinguishing non-linearly separable data using the perceptron rule for linear threshold units.

The extention ideas are based on two observations: i) when the perceptron cycles among hyperplanes which do not fully separate the patterns (when the problem is not linearly separable), a best split (minimum classification error) can be chosen and ii) it is always possible for the perceptron to build a hyperplane that separates at least one example from all the rest

The algorithm looks for the best hyperplane relative to the examples in the training set. If the problem is not completely solved by this best hyperplane, a new unit is added. This unit is connected to the inputs and to all the previous units. Thus, the dimension of the problem's space is extended and the problem could become linearly separable. In the worst case, each unit will classify a single pattern but experiments showed that this doesn't happen unless the problem is pathologically difficult (e.g. the two-spiral problem).

The divide and conquer networks [Romaniuk, 1993] use a divide and conquer approach to build a multilayer architecture similar to the cascade correlation one. The conquer stage is to add a first unit on a new layer. The unit is trained on all examples. If the unit does not classify correctly any training instance, a pattern is removed at random and the training is continued with the same unit. The process continues until at least one pattern is correctly classified. Then, the network enters the divide phase. In the divide phase, all the patterns which have been correctly classified are removed from the training set. The reduced training set is augmented with the closest neighbours of the set's patterns. After a (user set) number of epochs, the training halts with a certain number of patterns being correctly classified. These correctly classified patterns (if any) are removed from the training set and new neighbours added. If no patterns have been correctly classified, a pattern chosen at random is eliminated from the training set. This continues until a new unit classifies correctly at least one pattern. If at this stage, the error has not been reduced by a minimum (user defined) quantity, the network enters the conquer



phase in which a unit is introduced on a new layer. The new unit can be connected to the hidden units on the previous layer only or to all the previous units in the unit sharing variation. The net will have a single output unit which will be able to classify correctly the patterns in the training set.

## **2.4. Some factors influencing the training**

### **2.4.1 Architectural issues**

As shown, for many training algorithms, training difficulty increases with the complexity of the architecture, in particular with the number of layers. If the network has more than one hidden layer, other problems such as the attenuation of the error signal appear. Lang and Witbrock in [Lang, 1988] states that *"Unfortunately, back-propagation learning generally slows down by an order of magnitude every time a layer is added to a network. This is because the error signal is attenuated each time it flows through a layer, and learning progress is therefore limited by the slow adaptation of units in the early layers of a multi-layer network."* Lang and Witbrock solve the problem by using short-cut connections between every layer and each consecutive layer. These connections allow the error to propagate more efficiently in the whole network.

Deciding the architecture of a multilayer perceptron for a given I/O training set is a problem in itself. Many algorithms work with a fixed architecture. Therefore, the correct architecture has to be chosen before training starts. If training fails it is not clear if this is because of an insufficient architecture or another cause. An approach to solving this problem is to start with an architecture which is likely to have sufficient units for training by having many more. However, if the architecture is too rich, the solution weight state will have neurons which provide unnecessary information and/or neurons which do not contribute to the solution [Sietsma, 1991] and a pruning stage is then necessary. From this point of view, network construction algorithms which add units as they are needed seem to be a better choice if they do not have problems themselves.

### **2.4.2 Dimensionality of the weight space**

The training problem can be posed as that of finding the minimum of an error surface over a weight space. Independently of the algorithm used, this search problem becomes more difficult as the number of dimensions of the weight space increases. This type of behaviour is common to many different fields. In dynamic

programming for instance, the explosion of the amount of work and storage necessary to solve a problem determined by an increase of the number of variables has determined Bellman to talk about a "curse of dimensionality" [Bellman, 1957], [Ravindran, 1991].

Wilensky and Neuhaus (Wilensky, 1990) report that even for a simple, linearly separable problem like discrimination between samples drawn from two  $N$  dimensional gaussians, the training time increases both with the number  $N$  of dimensions for the same architecture and with the number of hidden units for the same dimensionality of the input space.

### **2.4.3 The pattern set**

Falhman and Lebiere in [Falhman, 1990] identified the moving target problem as one of the causes of the training problems for a multi-layer architecture trained using backpropagation. This effect is determined by the fact that all the units in the network evolve at the same time. Thus, the hidden units perceive a constantly moving picture and their task of evolving into useful feature detectors becomes far more difficult.

A manifestation of the moving target effect is the herd effect. The problem is determined by the presence in the training set of many different tasks to be accomplished. In this situation, more than one hidden unit will try to tackle the same task. Only after one of the tasks is accomplished by one or more hidden neurons, will other neurons be redirected to other sources of error.

The moving target effect and its particular manifestation as a herd effect, are some of the reasons for which the standard training is slow.

A different approach to the same problem describes the cause of the phenomenon as the credit assignment problem. According to this point of view, the problem is to assign each hidden unit to a particular feature to be detected. The problem is that in general there is no method to identify the features useful in solving a given problem. Furthermore, even the number of such features cannot be accurately estimated and thus the architecture of the network is often decided using rules of thumb and heuristics more or less precise.

#### **2.4.4 Conclusion. An ideal training problem.**

Let us try to sketch a general overview of the issues raised by a classification problem. The perceptron learning algorithm has an important positive feature: it is guaranteed to converge in a finite number of steps if the problem admits a solution i.e. if the patterns are linearly separable. As one moves to more complicated architectures/problems, this positive feature is lost from the associated learning algorithms. As shown, for many algorithms the training is affected by the number of layers, the dimensionality of the weight space, the dimensionality of the input space and the number of patterns in the training set. An increase in any of these parameters determines a more difficult training. The difficulty can manifest as an increase in the percentage of failures, a longer training in terms of number of epochs or simply in extra computational expense.

Let us try to define an ideal training situation by taking into consideration each of the factors discussed above. In order to avoid the attenuation of the error signal, the network should have as few layers as possible. The ideal from this point of view is to have just one layer. In order to keep the dimensionality of the weight space to a minimum, the network should have as few weights as possible. However, for a fully interconnected network the number of weights is determined by the number of neurons so the ideal training situation should involve training just one neuron at a time. The dimensionality of the input space depends on the problem and therefore cannot be changed to improve the training but the number of patterns in the training set can. Actually, the problem is not the number of patterns in the training set but the number of untrained patterns i.e. the number of sources of error. An ideal training situation would be such that the pattern set contain just one source of error (just one misclassified pattern).

Furthermore, an ideal training should be guaranteed to find a solution in a finite time and should be able to train architectures able to solve sufficiently complicated problems.

Other ideal training situations can be defined if the factors taken into consideration are different. For instance, if the error is not backpropagated through the network, the attenuation of the error signal is not a problem anymore and therefore, the number of layers is not directly important.

In conclusion, from the point of view of the factors discussed, an ideal training method is to train just one layer, just one neuron and to have just one source of

error in the training set i.e. just one pattern to learn. On the other hand, one needs architectures with more than one layer in order to solve non-linearly separable problems, quite a few weights which in the end are the degrees of freedom of the model used by the network and usually, the pattern set contains quite a few patterns, as well.

Most of the training methods briefly discussed above evolve in a training set-up very different from the ideal training situation as defined. Thus, backpropagation and all its variations try to train the whole network at a time with the complete pattern set. Independently of the particular training method used, the dimensionality of the weight space, the credit assignment problem and the attenuation due to the number of layers are problems which affect the performances of these training algorithms. If the derivative is estimated by perturbation, the number of layers remains important just because it affects the computation to be performed. The dimensionality of the weight space and the number of patterns to be learnt also maintain their importance.

For methods such as direct update of the weights and genetic algorithms, the dimensionality of the weight space becomes overwhelming because both techniques involve an extensive search of the weight space as opposed to a descent on an error surface. Although none of the techniques perform an exhaustive search of the weight space, the number of possibilities to be explored suffers an explosion as the number of dimensions of the weight space increases. Furthermore, neither error surface methods nor direct update of the weights or genetic algorithms have a guaranteed convergence.

Other classes of training algorithm such as radial basis functions and decision tree conversion offer an implicit solution to the credit assignment problem, the architecture being decided during the training. As these algorithms do not perform an explicit search in the weight space, the dimensionality of the solution weight space cannot be taken as the dimensionality of the search space. For these algorithms, the structure, the dimension and the content of the I/O pattern set become very important because the solution is determined by performing various calculations directly on the I/O pattern set.

Query learning falls into a separate category. Its training performance depends very much on the training set. Furthermore, this particular technique is based on the existence of an oracle able to give the correct output for any input pattern the technique might require and such an oracle might not be available in many

situations. However, even in those situations in which such an oracle is available, the convergence of the query learning algorithm is not guaranteed.

The constructive algorithms form the class of training algorithm which come closest to the ideal training situation as defined above. Thus, many of the constructive algorithms (upstart [Frean, 1990], tiling [Mezard, 1989], cascade correlation [Fahlman, 1990], etc.) train only a small part of the network at a time. The divide and conquer networks [Romaniuk, 1993] and Marchand's algorithm [Marchand, 1990] combine the training of a part of the network with some management of the training set which facilitates the training.

Although many of the constructive algorithms manage to combine some of the characteristics of the ideal training situation as defined, none of them have all the features of this ideal training. Some of them rely on a "best" partial solution being yielded by the perceptron training in those situations in which a proper solution does not exist. Others use a convergence process whose properties are guaranteed only for an infinite time. See for instance the pocket algorithm whose probability of finding the best solution increases to 1 only as the training time goes to infinity. At the same time, many of the constructive algorithms reviewed here need some heavy computation (building and converting classification trees for instance).

## **2.5. Assessing training speed**

### **2.5.1 Assessing the speed of one trial**

#### **Units of time**

In physics, speed is defined as the distance passed over per unit of time. The average speed during a given interval of time is defined as the quotient of the distance travelled during this time interval and the length of the time interval and the instantaneous speed is defined as the limit of the average speed when the length of the interval tends towards zero. In physics, speed is measured in units of length per units of time: meters per second, miles per hour, kilometres per hour, etc.

If the definition used in physics is to be extended directly to training neural networks, the definition of the training speed could be the number of patterns learned per unit of time measured in patterns per unit of time.

Unfortunately, no two problems are the same and it is very difficult to compare the complexity of two different problems. Then, perhaps the problem can be clearly

stated and two algorithms could be compared by running them on the same problem. In this case the speed would be measured in problems per unit of time where there is only one problem and this problem is clearly stated. In these conditions one has only to quote the time in order to give a measure of the speed. If a certain algorithm trains XOR in 100 seconds and another does it in 50 seconds, it is clear that the second one is twice as fast as the first one. Or is it?

Unfortunately, the algorithms are executed on machines and different machines have different performances. A comparison between the time necessary for a particular algorithm to train a particular problem is meaningful only if the machine is clearly specified. Furthermore, for a precise comparison one should state the compiler and the operating system, to say nothing about the programmer's skills. Clearly this is very unsatisfactory. Nevertheless, this method of reporting the speed performances of an algorithm by quoting only the time necessary to train a given problem (and the machine) is used very often. [Baum, 1991], [Baba, 1989], [Musavi, 1992], [Weymaere, 1991], [Brent, 1991] are just a few references in which the speed is reported in this way. Even if all the important factors were stated, this approach is still inconvenient because it requires the reproduction of the same experimental conditions for a meaningful comparison between different training algorithms. Thus, if a new algorithm A, tested on a new machine M is to be compared with set of existing algorithms  $A_1, A_2, \dots, A_k$  tested on machines  $M_1, M_2, \dots, M_k$  there are two possibilities for performing this comparison. A possibility is to implement the algorithms  $A_1, A_2, \dots, A_k$  on the machine M and the second possibility is to implement the algorithm A on the machines  $M_1, M_2, \dots, M_k$ . Both possibilities require a total of  $k+1$  implementations which is extremely inconvenient.

Sometimes, the training time is quoted more with the purpose of illustrating the effect of varying some parameters than offering a comparison with other techniques as in [Romaniuk, 1993], [Baba, 1989], [Wilensky, 1990] for example.

### **Epochs, pattern presentations**

A measure of the speed which is independent of the machine the algorithm is run on would be very useful. In looking for such a measure, one could consider the fact that a particular algorithm started with the same data should end up with the same result no matter the machine it is run on and this will be done by performing the same number of operations. If the convergence process uses a cycle, the cycle will

be performed the same number of times. In the framework of training, the most natural choice is the cycle over the training set.

The epoch, defined as a single presentation of the entire training set is a possible measurement unit for the training speed. However, there are algorithms which do not cycle through the pattern set or situations in which the size of the patterns set varies during the training. In order to cope with these situations, one could use the pattern presentation as a speed measurement unit. For those techniques which use a fixed, finite training set one can easily calculate the training speed in epochs if the same is given in pattern presentations or reciprocally. This is the measure proposed by Falhman in [Falhman, 1988].

### **Connection-crossings**

Comparison of training algorithms which use the number of epochs or pattern presentations as a machine independent measure are appropriate only when the algorithms being compared involve similar amount of work per epoch or pattern presentation [Brent, 1991].

There are algorithms which need to propagate the error only in a limited part of the network and/or in only one direction. This is the reason Falhman in [Falhman, 1990] proposes the number of connection-crossings as a measure of the learning time and implicitly of the learning speed. According to Falhman, the learning time measured in connection-crossings is the number of multiply-accumulative steps necessary to propagate activation values forward through the network and error values backward.

This is appropriate for those algorithms which perform operations of the same computational load for each connection. However, there are algorithms which do different things. Some algorithms might converge in very few epochs, propagate values only forward through the network and therefore have few connection-crossings. However, the same algorithm could need the calculation of the inverse of a large matrix for each of the connection-crossings for instance, and thus it could require a lot of CPU time.

### **Number of operations**

A better measure for the training speed which can be applied to virtually all types of training algorithms is the number of operations as defined in the field of algorithmic analysis and design. This can be done by counting certain operations (the ones

which are estimated to take a significant time) and expressing the performance modulo some multiplicative constant.

As opposed to the algorithm analysis in which the performance is associated with the algorithm, in the assessment of the speed of a training session the number of operations is used to give an indication of the computation involved in that particular training session. For instance, Brent in [Brent, 1991] uses the number of operations to characterise the general performance of the algorithm and to compare it with standard backpropagation. In doing this, he is forced to make some assumptions about the problem (a generic problem) which cannot be sustained by theoretical reasons. Later on, when reporting the performance, Brent uses the more common, but less informative training time in seconds on a particular machine.

The approach presented in this thesis, proposes using the number of operations both in assessing the performance of the algorithm and in reporting trial results. Eventually, the latter could sustain some assumptions used by the former.

Let us consider the example of an algorithm which performs a global operation on the weight matrix. Let us assume this operation needs  $w \cdot \log(w)$  operations where  $w$  is the number of weights in the network (the architecture is fixed). This operation is performed for each pattern presentation. Let us suppose this algorithm uses a training set which increases linearly so that the first training set contains 1 pattern, the second 2 patterns and so on. Furthermore, this algorithm uses  $k$  passes through the network for each iteration and the total number of patterns is  $n$ . In a particular case in which  $e$  epochs (an epoch is defined as a presentation of the entire current training set) are necessary for each training set, the training time could be expressed as:

$$\begin{aligned} e[kw \log w + 2kw \log w + \dots + nkw \log w] = \\ = e \sum_{i=1}^n i kw \log w = ekw \log w \sum_{i=1}^n i = ekw \log w \frac{n(n+1)}{2} \end{aligned} \quad (1)$$

This shows how this measure can be used to characterise the performance of a given trial independently of the machine it is used on and in the situation in which the algorithm's processing is non-standard. Any other performance measure would be misleading if applied to this algorithm.

In the case of standard backpropagation, quickprop and cascade correlation, this measure reduces to the connection-crossing measure because there are no complex



operations associated with a connection and there are no global operations on the weight matrix or pattern set. Thus, the number of operations measure would give the same comparison between backpropagation, quickprop and cascade correlation as the one given by the connection crossings measure. For instance, the value of this measure for the backpropagation is  $w*2*n*e$  where  $w$  is the number of weights, 2 is the number of passes through the weights for a pattern presentation,  $n$  is the number of patterns and  $e$  is the number of epochs. A comparison is now possible between backpropagation and the hypothetical algorithm considered before.

In those cases in which the training depends very much on the problem and perhaps the initial state, this measure cannot be estimated a priori as a function of the various parameters but can be easily reported by the implementation of the algorithm.

### **2.5.2 Assessing the average learning speed**

As the result of a particular trial can depend on factors like initial weight state, more reliable information is obtained by calculating some mean values over a number of trials and some measure of the variation of individual cases. Falhman in [Falhman, 1988] discusses some issues regarding the assessment of the speed in the case in which more than one trial is considered.

If a number of trials are used, the problem of how to treat the failures must be analysed. Some algorithms can become stuck in some particular states or can have anomalous long training times (which are effectively infinite). How can one calculate the mean value of a series of values which can include infinite values? For the purpose of this discussion, the units used for the measurement of individual trials are not relevant. The individual trials can be expressed in number of epochs, pattern presentations, connection crossings or operations and the average will be expressed in the same units. In the following, 'units' can be substituted with any of the above.

A possibility is to report separately the successes and the failures for each set of trials as in [Moeller, 1993], [Weir, 1991] and others. As Falhman points out, if we do this, how will we choose between a technique with a better average and one with fewer failures?

An approach proposed by Tesauro and Janssens is to define the training rate to be the inverse of the time required for that trial. An average training rate can be

calculated and the average training time is defined by the inverse of this average training rate. Fahlman criticises this approach as i) penalising more consistent algorithms, ii) favouring algorithms which combine taking risky steps for a very short training with many failures and iii) emphasising short trials with respect to long ones. Let us analyse this.

According to this method, the average training time (which can be seen as a training cost measured in units) is calculated as:

$$\bar{c} = \frac{n}{\sum_{i=1}^n \frac{1}{c_i}} \quad (2)$$

where  $c_i$  is the cost of one trial,  $n$  is the number of trials and  $\bar{c}$  is the average cost of training. The terms in equation (2) can be split into terms corresponding to successful trials and terms corresponding to unsuccessful ones:

$$\bar{c} = \frac{n_s + n_f}{\sum_{i=1}^{n_s} \frac{1}{cs_i} + \sum_{i=1}^{n_f} \frac{1}{cf_i}} \quad (3)$$

where  $cs$  are the costs of successful trials,  $cf$  the costs of unsuccessful ones,  $n_s$  the number of successful trials and  $n_f$  the number of unsuccessful ones. If the number of failures is 0, the average value calculated according to this method is different from the arithmetical mean and this is why the more consistent functions are penalised i.e. they would appear to have a slower training.

Fahlman's idea was to restart the training, with random weights, whenever the network has failed to converge after a certain number of epochs. The duration of a trial is the total number of units since the previous successful trial. This approach offers the advantage of giving the arithmetical mean if there are no failures, thus eliminating the bias of the previous method. However, this approach has a drawback: the implicit dependence on the termination limit i.e. the number of units after which a training is restarted. The same algorithm could give different values for the average convergence time if the termination limit is taken to be different (and the algorithm fails from time to time). Furthermore, this termination limit depends on the algorithm itself. It is not reasonable to use the same limit for two algorithms which usually converge in 20 and 2000 units respectively.

A solution is to calculate a cost of training in which the cost of a normal training session and the cost of a failure are evaluated taking into consideration the particular requirements of the problem. The cost of a training in units is defined as:

$$\text{cost}_{\text{total}} = \text{cost}_{\text{succes}} + \text{cost}_{\text{failure}}$$

$$\text{cost}_{\text{total}} = \sum_{i=1}^{n_s} \text{cs}_i + \text{cs}_{\text{failure}} n_f$$

$$\bar{c} = \text{cost}_{\text{average}} = \frac{\text{cost}_{\text{total}}}{n_s} = \frac{\sum_{i=1}^{n_s} \text{cs}_i + \text{cs}_{\text{failure}} n_f}{n_s} = \frac{\sum_{i=1}^{n_s} \text{cs}_i}{n_s} + \frac{\text{cs}_{\text{failure}} n_f}{n_s} = \text{cost}_{\text{succ}} + \text{cs}_{\text{failure}} \frac{n_f}{n_s} \quad (4)$$

In this formula,  $\text{cs}_i$  are the costs of successful trials,  $\text{cf}_i$  the costs of unsuccessful ones,  $n_s$  the number of successful trials,  $n_f$  the number of unsuccessful ones,  $\text{cost}_{\text{succ}}$  is the average cost of the successful trials and  $\text{cs}_{\text{failure}}$  is the cost of a single failure. If the number of failures is 0, this formula yields the value of the arithmetical mean (the average cost of successful trials).

If the cost of a failure  $\text{cs}_{\text{failure}}$  is taken to be the termination limit  $N_{\text{max}}$ , the formula (4) models the strategy adopted by Falhman which can be shown by rewriting (4) as:

$$\bar{c} = \frac{\sum_{i=1}^{n_s} \text{cs}_i + N_{\text{max}} n_f}{n_s} = \frac{\sum_{i=1}^{n_f} (\text{cs}_i + N_{\text{max}}) + \sum_{i=n_f+1}^{n_s} \text{cs}_i}{n_s} \quad (5)$$

The last form of expression (5) uses two assumptions: i) that the number of failures is less than the number of successes so that each failure can be coupled with a success and ii) that the failures have occurred one before each of the first  $n_f$  trials. These assumptions are not essential and do not modify the result (due to the commutativity of the sum).

If these costs are correctly evaluated for a given problem, a comparison can be performed between any two algorithms even if one of them is guaranteed to converge and the other can fail. The dependence on the termination limit is present but it is made explicit and can be controlled precisely. Furthermore, this approach allows making a better choice of the training algorithm in those situations in which the training is performed on-line and a failure can be more costly than simply the number of iterations wasted.

# CHAPTER 3

## On "good" generalisation

### 3.1. Introduction.

#### 3.1.1 A definition of generalisation for feedforward networks.

In a symbolic artificial intelligence framework, generalisation can be defined as: "*to take into account a large number of specific observations, then to extract and retain the important common features that characterise classes of these observations*" [Mitchell, 1982]. This definition seems more appropriate to describe concept learning than generalisation. Indeed, a system which has extracted and retained the common features of a set of examples can be said to have learned the concept embedded into those patterns. As long as the system does not try to use this knowledge upon untrained examples, it cannot be said to generalise.

Rumelhart and McClelland in [Rumelhart, 1986] talk about generalisation in the following terms: "*The fact that similar patterns tend to produce similar effects allows distributed models to exhibit a kind of spontaneous generalisation, extending behaviour appropriate for one pattern to other similar patterns. (...)*".

One can reformulate this statement: the fact that inputs close together in input space tend to produce outputs close together in output space, allows distributed models to exhibit a kind of spontaneous generalisation, extending behaviour appropriate for one pattern to other similar patterns (i.e. giving for an unseen input, an output close to the output of nearby trained inputs). This definition contains Mitchell's idea of "common features" in the similarity of effects determined by similar patterns. Furthermore, Rumelhart's definition introduces the idea of untrained inputs. The system is said to generalise when it acts upon untrained inputs.

In a mathematical language, the similarity of effect determined by similar inputs is continuity of the input/output function: to an infinitesimal change in the input, there corresponds an infinitesimal change in the output.

However, this definition seems also to characterise a desirable feature. In other words this could be a definition not for generalisation but for good generalisation.

Sietsma defines generalisation as "the ability to recognise or correctly classify patterns which have never been presented to the network before" [Sietsma, 1991]. Once more, the definition contains a desirable element: "correctly". According to this definition, a system which classifies incorrectly an unseen pattern does not generalise at all for that particular pattern.

Baum considers the problem of learning in the following terms. A set of examples is stored in a system (a neural network for instance). Subsequently, the system is challenged with some other examples drawn from the same distribution. The system exhibits a valid generalisation if the output is correctly predicted. The characteristic of this approach is that it does not assume the existence of an underlying function. Indeed, the underlying process that generates the examples need not be a function and can classify them in a stochastic manner [Baum, 1989].

Denker considers a universe  $U$  of relations and the memorisation data  $M$ , a subset of  $U$ . A generalisation of  $M$  is any subset  $G_i$  of  $U$  so that  $M$  is a proper subset of  $G_i$ . In other words, the relation  $G_i$  has a larger domain than  $M$  and the two relations agree wherever their domains overlap. It is emphasised that many possible generalisations exist for any memorisation data  $M$  [Denker, 1987].

Poggio and Girosi present the learning by example problem as a problem of approximating a multivariate function [Poggio, 1990; Girosi, 1990]. In this approach, learning means collecting the examples, that is input/output pairs and generalisation means estimating the output at locations in the input space where there are no examples. In this sense, learning and generalising is a problem of hypersurface reconstruction.

Sometimes, generalisation is defined in a framework of associating input contours in the plane to output contours in the plane [Takahashi, 1993]. The contours are parametrised by a variable  $t$  taking values in an interval  $I$  which is partitioned into subintervals  $I_k$ . A generalisation (more precisely a  $\epsilon$ -generalisation) is an input/output mapping such that the distance between this mapping and the training value is less than a small positive value  $\epsilon$  for the corresponding subinterval  $I_k$ . Conversely, given an I/O mapping, one can obtain a training set for it by simply taking samples so that the above distance condition is satisfied.

As shown, there are many possible approaches to defining generalisation. Each of them is more or less convenient to particular methods of analysis. In the following,

it will be assumed that the training set is finite and the training patterns specify the outputs corresponding to given inputs.

**Definition.** Generalisation is the property of a feedforward neural network to give outputs for inputs which are not in the training set. The word generalisation will also designate the I/O mapping obtained by associating the inputs with the targets for the inputs in the training set and with outputs obtained using the generalisation property for those input points which are not in the training set.

**Observation.** The definition assumes that there are no one-many associations in the training set (otherwise, the I/O relation including the training set is not a function).

This definition is consistent with many of the approaches presented above. Rumelhart's and Sietsma's descriptions can be seen as describing good generalisation and so can Baum's "valid generalisation" term. If the universe  $U$  of relations is defined as the set of pairs (input, output) where the output is generated by the net when some input is presented, the above definition is consistent with Denker's framework as well (Denker's is more general because there are relations which are not functions). At the same time, this definition fits very well with Poggio's approach of surface reconstruction. Takahashi's definition is restricted to mappings from a 2D space onto another 2D space but is otherwise compatible with the above definition because it assumes the net gives some outputs for inputs different from those in the training set.

There is an observation to be made. This definition does not differentiate interpolation from extrapolation. While interpolation is well studied and understood, the extrapolation is far less under control. There are many effective interpolation techniques but they are almost always poor at extrapolation. The distinction between the situations in which the network is asked to interpolate and to extrapolate can be quite useful and, as shown in section 5 of this chapter, meaningful for assessing the correctness of the output when the network generalises.

According to the definition above, any neural net is able to generalise because any neural net, trained on any training set will give some output if tested with an untrained pattern. The question is whether this output is useful or not. Can we accept Sietsma's definition as the definition of good generalisation? What is good generalisation when the net is not a classifier and how do we assess it?

These issues will be discussed in the rest of this chapter. The topic of the discussion is how to assess generalisation not how to obtain good generalisation. Thus, approaches like Hinton's bottleneck and other specific techniques designed to ensure good generalisation will not be discussed.

### **3.1.2 Common methods for assessing generalisation.**

One commonly says that the generalisation is good if the output for some inputs outside the training set is close to one's own expectations. If the net is used to model an existent system or phenomenon, these expectations are given by the output of the system to be modelled. If such a system doesn't exist, the expectations are usually the result of one owns interpolations or extrapolations.

Let us consider the I/O patterns in fig. 1 for instance. One looks at the points and gets the image of a sine function. If the network's outputs for inputs between the inputs in the training set are close to those given by a sine function, one says that the generalisation is 'good'.



Fig. 1. I/O patterns.

## **3.2. Are there enough patterns in the training set?**

### **3.2.1 An approximation point of view.**

Let us consider a net used to model a real system. The system's transfer function is sampled and the samples are the I/O patterns in fig. 2. One looks at the points and gets the image of a straight line. A net having a straight line as its overall transfer function will be considered as having good generalisation.

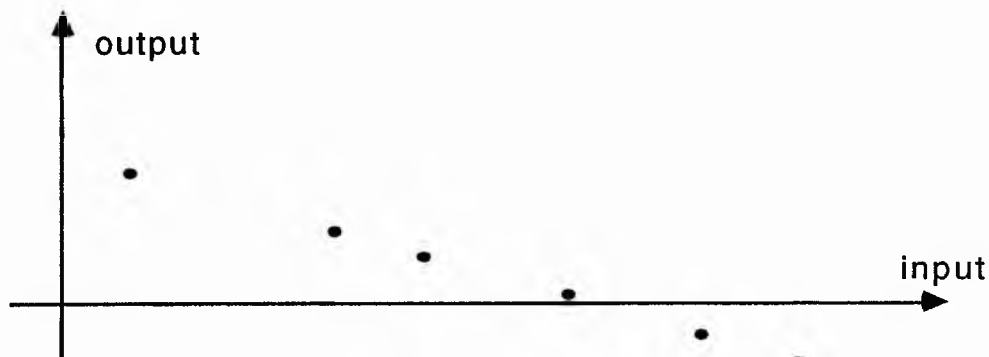


Fig. 2. The observer interpolates the I/O patterns to obtain an expected generalisation which will be used to assess the generalisation of the net. In this case, the observer's expected generalisation is a straight line.

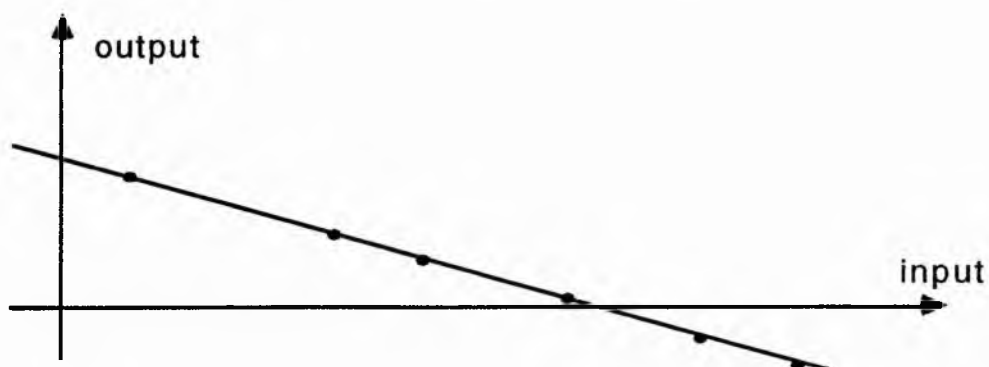


Fig. 3. A possible transfer function of a net modelling the system which produced the samples in fig. 2. The net will be judged as giving "good" generalisation.

Now, the user of the system compares the net's model (the straight line) with the real transfer function of the net presented in fig. 4 and declares their disappointment. A neural net person defending the net could say that it didn't have enough samples to model that transfer function. The user of the real system could reply that the transfer function was just a sin and there were 6 samples for 3 periods which is the minimum number of samples requested by a Fourier technique, for instance. Thus, the problem of the number of patterns in the training set (i.e. samples) arises. If we know the function to be modelled, how many samples do we need? And if we don't know the function and we are limited to a finite set of samples as in the example above, what sort of confidence can we have that the model will reflect the real properties of the underlying function?



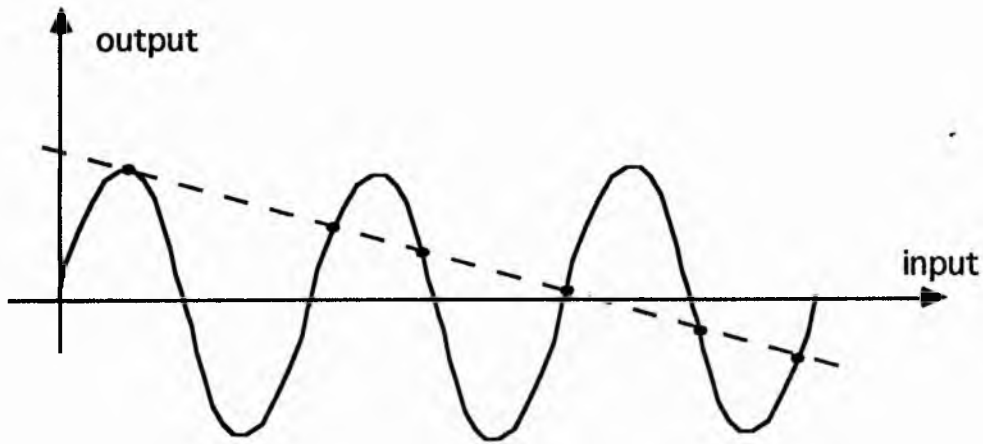


Fig. 4. The samples in fig. 2 were samples from a sine function. The 'good' generalisation of the net proves to be a very poor one.

One could say that in a neural net framework the number of samples is large enough only when the samples 'sketch' the shape of the function. In this case, what does 'sketch' mean? Does one have a valid sketch when the shape of the underlying function is 'the same' as the shape of the function obtained by linear interpolation between the samples? If so, there are curves which can never be sketched no matter the number of samples, for instance the Mandelbrot set  $z^2 - \lambda$  (fig 5). No matter the number of samples one uses, the curve will never be correctly "sketched" because between any two sample points there is another structure of infinite complexity. In order to appreciate this, one has only to change the scale of the exploration.

Does one have a valid sketch when the difference between the linear interpolation of the samples and the underlying function is below an error limit at any point? If so the Mandelbrot set would not be a problem.



Fig. 5. No matter the number of samples, the shape of the function obtained by linear interpolation will be different from the shape of the underlying function.

The answer to this question depends very much on the application. The following example has been used by W.W. Sawyer in [Sawyer, 1966].

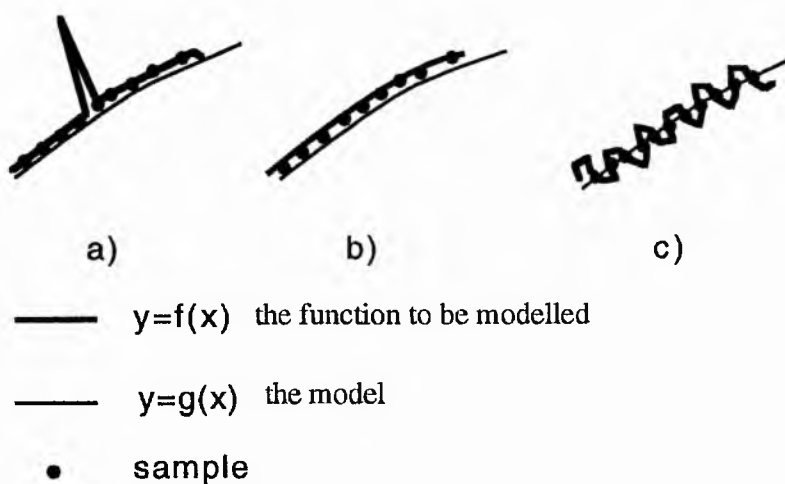


Fig. 6 The underlying function  $y=f(x)$  is sampled. From these samples the function  $y=g(x)$  can be obtained through linear interpolation. Are there enough samples?

In fig. 6, in each of diagrams (a), (b), (c), we see an underlying function  $y=f(x)$ . This function is sampled and an interpolation  $y=g(x)$  is obtained. The question is whether the number of samples is enough to characterise the underlying function  $y=f(x)$ . The answer depends very much on one's purpose. In (a),  $y=g(x)$  goes a long way from  $y=f(x)$  but it only stays away for a very short time. If we were mainly interested in the areas under the two curves, it might well be that these areas would differ by very little so  $y=g(x)$  is a good approximation and the number of samples was sufficient. With this criterion, the curves in (b) and (c) would be close together as well and therefore, the number of samples would have been sufficiently large even in these cases. However, it might be that we want to ensure that the difference between the underlying function and its approximation is not greater than a given error limit for any  $x$  value. In these case, the number of samples is insufficient for the function in (a) but is still sufficient for the functions in (b) and (c). In an investigation where we are particularly concerned with the length of the curves the number of samples would be seen as insufficient for both functions in (a) and (c).

The conclusion of this example is that the number of samples itself cannot be declared as sufficient or insufficient independently of the particular problem or type of problem. On the other hand, once the type of problem has been stated - and only in these conditions - one can assess the fitness of the sample set.

In this context, a problem is defined as a triplet consisting of a training set (an I/O problem), an error measure and an error limit. The error measure and the error limit depend on the application. The same I/O problem can be part of two different problems if the error measure or the error limit is different.

### **3.2.2 A neural network point of view.**

The decision whether a training set contains enough patterns or not depends on the type of network as well. In this context, the type of a network is given by its intrinsic mechanism. Some such mechanisms are:

#### **1. The classical linear network:**

$$F(W, X) = W^T X \quad (1)$$

where  $W$  is a weight vector and  $X$  is a input vector. This corresponds to a network without hidden units.

## 2. The basis functions network:

$$F(W, X) = \sum_{i=1}^m W_i \Phi_i(X) \quad (2)$$

where  $\Phi_i$  are some function which form a basis. This corresponds to a radial basis function network for instance.

## 3. The multilayer sigmoid network:

$$F(W, X) = \sigma \left( \sum_{i=1}^m w_i \sigma \left( \dots \sigma \left( \sum_{j=1}^n w_j x_j \right) \right) \right) \quad (3)$$

where  $\sigma$  is a sigmoidal function.

It is assumed that each type can use as many units as necessary for the training to be successful.

Let us suppose that the problem is to model a linear function in a 2-dimensional space (one input, one output). If the net uses a linear mechanism, 2 samples on the hyperplane are enough for the net to be able to build its best representation of the problem. If the net uses a radial basis mechanism with hyperspherical basis functions, a number of samples equal to the number of hidden units will be necessary for the net to build its best possible representation. The linear net's model will be much better than the radial net's one but this is not relevant for the problem. The important aspect is that different types of nets need different numbers of samples (training points) to build up the best representation (of the given underlying function) they are able to. For this argument, the particular method used to compare different representations is not important as long as it is the same for all comparisons.

### 3.3. Good generalisation.

#### 3.3.1 Training without validation set.

Let us consider the common situation in which the only information available is the training set (i.e. the function to be modelled is unknown) and the following example. Three different nets are trained with the given training set and each of them finds a different final weight state i.e. a different transfer function as in fig. 7. There is no reason to consider one of these functions as being better or worse than

any of the other two or than any other function passing through the given training points. In this situation, good (or bad) generalisation doesn't exist.

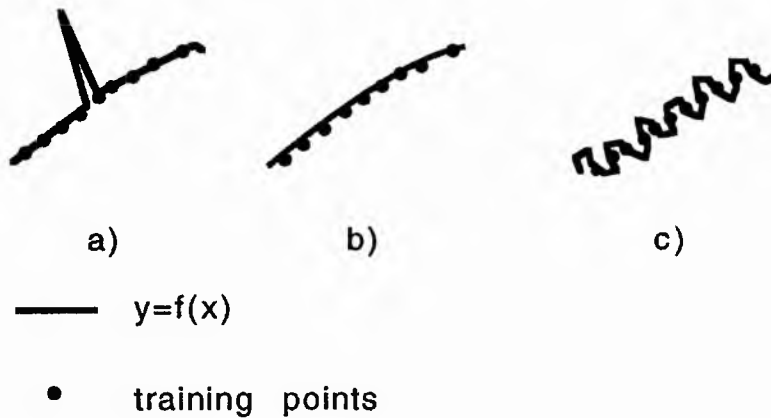


Fig. 7 The training points are the only information available. Three different generalisation are presented. In this situation, none of them is better or worse than the other.

This approach is sustained by Poggio and Girosi in [Poggio, 1990]. Learning from examples is seen as a surface reconstruction process. No particular surface is seen as the desired surface (the "good" generalisation) and the need for assumptions is made explicit.

"From this point of view, learning a [...] mapping from examples is clearly an ill-posed problem, in the sense that the information in the data is not sufficient to reconstruct uniquely the mapping in the regions where data are not available. In addition, the data are usually noisy. A priori assumptions about the mapping are needed to make the problem well-posed. One of the simplest assumptions is that the mapping is smooth: small changes in inputs cause a small change in the output. [...] Other stronger a priori constraints may be known before approximating a mapping, for instance that the mapping is linear, or has a positive range, or a limited domain or is invariant to some group of transformations".

Although the definition of the generalisation can be different, the same need for assumptions is present in the symbolic AI world. Mitchell in [Mitchell, 1980] shows the need for biases of the generalisation language: "generalizing from a small set of training instances is possible only under a priori biases for choosing an appropriate generalization out of the many possible".

This point of view according to which any generalisation is as good as any other seems incompatible with the approaches which try to offer guarantees regarding generalisation. [Hertz, 1990] presents an approach originally introduced by Schwartz, Denker et.al. [Denker, 1987; Schwartz, 1990] which offers some results regarding the average generalisation ability. The fundamental idea is to consider the volume of the weight space compatible with the given training set and with the goal function, and the total volume of the weight space. Their result is based on cardinality and entropy and gives an estimation of the number of examples one must use to ensure a generalisation error less than a given error limit.

Baum et al in [Baum, 1989] calculates lower and upper bounds on the sample size as a function of the error limit  $\epsilon$  and the net size (number of nodes  $M$  and number of weights  $W$ ) needed such that valid generalisation can be expected. If the error on the training set is less than  $\epsilon/2$ , at most (the upper bound) of the order of  $\frac{W}{\epsilon} \log \frac{M}{\epsilon}$  examples are needed to obtain a generalisation error less than  $\epsilon$ . Their result applies to networks with linear threshold functions (or at least binary outputs) and is based on the notion of capacity. The particular measures of capacity used in this paper are the maximum number of dichotomies that can be induced on  $m$  inputs and the Vapnik-Chervonekis dimension.

As stated, these results come into an apparent contradiction with Poggio and Girosi's approach: how could one offer guarantees regarding good generalisation if any generalisation is as good as any other?

In reality, the contradiction does not exist because Schwartz's and Baum's results do not make any assumptions about any particular good generalisation. They simply ensure that no matter which generalisation is the correct one, the difference between it and the function actually implemented will be smaller than the error limit. An example of the same strategy could be betting on a horse. We could bound the loss to  $e$  by betting  $e/(n-1)$  pounds on each of  $n$  horses. No matter which horse is the winning one, we cannot lose more than  $(n-1) \frac{e}{n-1} = e$  pounds. The actual loss will be less than  $e$  because the  $n$ -th betting on the winner will bring some money. It is noticeable that one could give this bound without knowing the winning horse, i.e. the "good" generalisation.

In the same way, if we have enough training patterns so that the difference between any two functions consistent with the training patterns is less than the error limit,

then we can be certain that no matter which function will be chosen as the good generalisation and which function will be implemented by the net at the end of the training, the generalisation error will be less than the same error limit.

Let us explain this. Let us suppose that the previous condition is satisfied i.e. the difference between any two functions consistent with the training patterns is less than the error limit. On the one hand, the function implemented by the net (the actual generalisation) is consistent with the training set because this was the termination condition for the training process. On the other hand, the good generalisation is one of the possible generalisations and all possible generalisations are consistent with the training set by definition. Both the good generalisation and the actual generalisation are consistent with the training set and therefore, the difference between them is less than the error limit.

### **3.3.2 The validation set.**

A common method to assess the performance of a net is to use a validation set. The idea of a validation set is to test the network on data which was not used during the training and to compare the output given by the net with the known output. Can such a test indicate when the net offers good generalisation?

The available data (assumed to be finite) is divided into two sets: the training set and the validation set. The training set is used to train the net (to actually change the weights) whereas the validation set is used to assess the performances of the trained network. If more than one solution (low error on the training set) is available, the one with the minimum error on the validation set will be chosen. This situation is to be compared with the situation in which the net is trained with all available data.

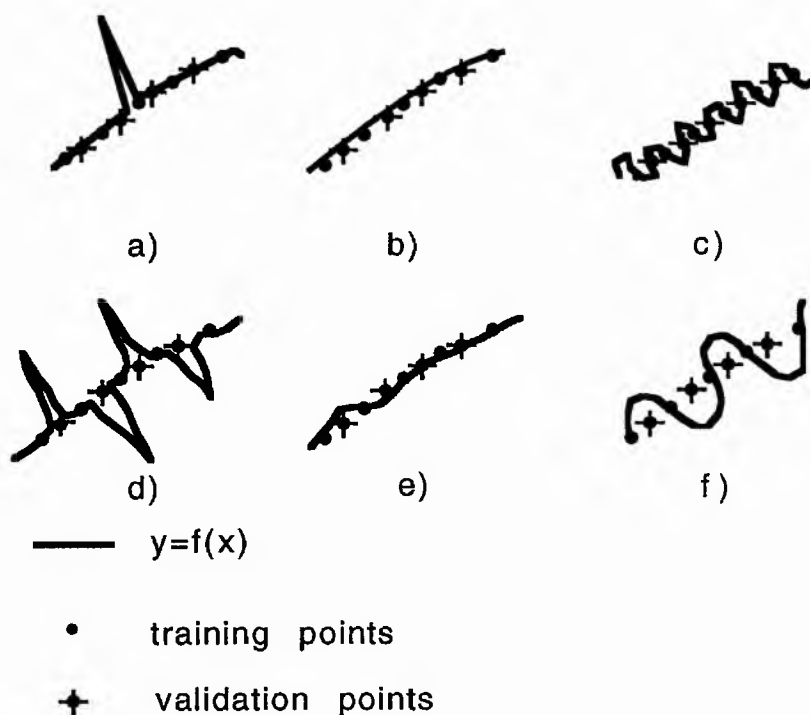


Fig. 8 The validation set will eliminate (d), (e) and (f) only. If the data in the validation set was used in training, (d), (e) and (f) would not be valid solutions.

The actual training set (as opposed to available data) contains fewer training points and less information is available to the part of the training algorithm which is responsible for the weight changes. As shown in [Schwartz, 1990], the volume of the weight space consistent with a number of instances is inversely proportional with the number of instances. Therefore, more solutions will be available. Let us suppose that there are six possible solutions as presented in the fig. 8 above.

Some of them will give poor results on the validation set and will be eliminated. However, none of the solutions which could have been found in the first case (training with all available data) will be eliminated. The problem of reducing the set of possible solutions has not been solved by the use of the validation set. There is still no reason to declare one of the functions in (a), (b), or (c) better or worse than any other (from (a), (b) or (c)). The validation set is able to eliminate only those solutions which would have never been solutions if the validation set had been used effectively in the training i.e. (d), (e), and (f).

The conclusion of the argument above is that for any finite number of samples, there are many functions which could pass through those points. Any of these functions is a valid generalisation. None of them is better or worse than any other.



A net implementing any one of them (i.e. passing through all available data points) will give a valid generalisation and the use of a validation set does not help the assessment of these possible generalisations.

Is the validation set completely useless then? If not, what is its use and how should it be used? The following section will attempt to answer these questions.

### **3.4. The use of the validation set**

For simplicity, the discussion will consider the case of a classifier but the same ideas are valid for other types of neural networks.

Let us consider an input space  $X$ , a set of classes  $C$  and a classifier  $d$  defined on  $X$  and taking values in  $C$ .  $R^*(d)$  will denote the true misclassification rate of the classifier  $d$ . The meaning of  $R^*(d)$  is the following: Using a sample population  $L$ , construct the classifier  $d$ . Draw another very large (virtually infinite) set of samples from the same population  $L$  and compare the prediction of the classifier  $d$  with the correct classification for each sample. The proportion of misclassifications given by  $d$  is the value of  $R^*(d)$ .

The size of the sample set is required to be sufficiently large so that the statistical techniques used and various probabilities are meaningful. If the size of the various sets (training, validation, etc.) is not large enough, the estimates given by the statistical techniques will be poor.

A more detailed framework for the true misclassification rate and various validation techniques can be found in [Breiman, 1993]. The most commonly used validation techniques will be briefly presented in the followings.

A distinction must be made between computer simulations and real world problems. In the case of a computer simulation, the true misclassification rate can be calculated using the definition. A random number generator is used to construct the sample set. Then, the classifier is built using these data. Another set of samples is obtained using the same distribution and the true misclassification rate can be easily calculated according to the definition.

The real world problems can be divided into two types: problems in which the amount of data is virtually infinite and problems in which the amount of data is finite. If the amount of data is very large, the estimate of the true misclassification rate can be calculated again according to the definition with the precaution that

independence of various pieces of data be assured. In many problems though, only a finite set of samples is available with reduced possibilities of getting an additional very large set of correctly classified samples. Due to the fact that the same data set is used both to construct and to validate the classifier, the estimate of the true misclassification rate  $R^*(d)$  is called an internal estimate.

#### **3.4.1 The resubstitution estimate.**

A common technique for calculating  $R^*(d)$  is the resubstitution technique. After the classifier  $d$  is constructed, the data used in its construction is fed to its input and the misclassification rate is calculated by comparing the classification given by the classifier to the real class of each input pattern. Thus, the resubstitution estimate is obtained.

The main disadvantage of this estimate of the misclassification rate is the fact that usually, the construction algorithm tends to minimise a value proportional to the difference between the output given by the classifier and the desired target and therefore proportional to the resubstitution estimate. Therefore, this estimate is bound to be overly optimistic. For instance, there are techniques which ensure that all the patterns in the training set are correctly classified. In this case, the resubstitution estimate of the true misclassification rate is zero. It is difficult to accept that this classifier will correctly classify all the patterns drawn from the same distribution.

In conclusion, the resubstitution estimate is not sufficiently accurate for most purposes. It reflects only how good the classifier construction algorithm is and says very little about how the resulting classifier will behave with new data. In a neural framework, a measure of the resubstitution estimate is the error at the end of the training and the dangers of using this as a measure of the performance on new data have been long understood.

#### **3.4.2 The test sample estimate.**

This method requires the division of the input pattern set into two sets usually called the training set  $T$  and the validation set  $V$ . The classifier is constructed using the samples in  $T$  and the misclassification estimate is calculated using  $V$ . This method offers the advantage of using independent samples to construct and to test the classifier. The most important drawback of this method is that the size of the training set is reduced. A frequent but not theoretically justified division of the

available data puts 2/3 of it in the training set and the remaining 1/3 in the validation set ([Breiman, 1993]). In some cases, this loss of information can dramatically affect the resulting classifier. However, the larger the data set, the lower the probability that the samples in the validation set (which are missing from the training set) be important for the construction of the classifier.

Another critical condition of this method is the need for the training set and the validation set to be drawn from the same distribution i.e. to reflect to the same extent the intrinsic properties of the phenomenon ([Breiman, 1993], [Denker, 1987]). For instance, if a function is to be approximated on an interval  $I$  and all the training samples are chosen from the first 2/3 of  $I$  and the validation samples are chosen from the last 1/3 of the interval, both the classifier and the test sample estimation can be very poor. The common method to ensure this representativity condition is satisfied is to construct the validation set by randomly choosing patterns from the available data.

In conclusion, two conditions are necessary for the test sample estimate to be accurate: i) a (very) large data set and ii) the training set and the validation set to reflect to the same extent the properties of the underlying phenomenon.

#### **3.4.3 The V-fold-cross-validation estimate.**

This method requires the division of the available data  $L$  into  $V$  sets of the same size (or as close as possible)  $L_1, L_2, \dots, L_V$ . The  $V$  test sample estimates are calculated, each time using the training set  $L - L_i$  ( $L$  minus  $L_i$ ) and the validation set  $L_i$ . The  $V$ -fold-cross-validation estimate is the arithmetical mean of the  $V$  test sample estimates. At the end, the classifier  $d$  is constructed using the entire data set  $L$ .

A variation of this validation method is the "leave-one-out" method. If the data set contains  $N$  patterns, one of the patterns will be ignored each time and a classifier will be constructed using the remaining  $N-1$  patterns. Then the ignored pattern will be used as a single-case test and  $R^*(d)$  calculated as the mean of the misclassification estimate for each of the  $N$  cases.

Cross-validation is parsimonious with data. Every sample is used to construct the classifier and every sample is used exactly once in a test sample. The main drawback of this approach is that the process is very tedious. The construction of  $V$  classifiers is required and each such construction can be difficult.

#### 3.4.4 Conclusions regarding the use of the validation set.

The validation set is useful to obtain an estimate for the true misclassification rate which is an indicator of the expected performance of the classifier in the normal use after training. The validation set will not be able to distinguish between different weight states which satisfy both the training set and the validation set. These indistinguishable weight states are those which have been obtained if the validation set had been included in the training set. However, the validation set will emphasise those weight states or training methods which are able to guess some points even if they are not present in the training set. This justifies hopes that the nature of the underlying phenomenon has been embedded into the model built by the network.

If the sample set is small (much smaller than Baum's upper bound of  $\frac{W}{\epsilon} \log \frac{M}{\epsilon}$ , for instance), each pattern is important and a cross-validation technique will offer the best results. After the misclassification estimate is calculated, the construction of a new classifier taking into consideration all available data can be performed. This should ensure a true misclassification rate not worse than the test sample estimate obtained in the first place.

This conclusion follows from the approach to generalisation presented in [Schwartz, 1990]. Their results show that the learning determines "a monotonic increase of the average generalisation ability with increasing  $m$ " where  $m$  is the number of patterns in the training set. Therefore, the training set must contain as many patterns as possible and each pattern removed from the training set will affect negatively the generalisation. This is why, the final phase of retraining the net with the whole training set is very important.

If the sample set is large or very large, the individual patterns are far less important. Amari shows in [Amari, 1993] that the average information gain (the average of the logarithm of the probability of correct classification of a new pattern after  $t$  patterns have been learned) converges to 0 as  $d/t$  where  $d$  is the number of modifiable parameters. If  $t$  is very large, the information gain brought by an individual pattern is very small and some of them can be taken out from the learning set without damaging the performance of the net. In this case, the test sample estimate is more feasible. Although potentially useful, the final training with all available data can be skipped in this case if the generalisation performance is ensured by some bounds (e.g. Baum's) or is declared satisfactory by the user.

However, in the case of the test sample estimate, precautions must be taken to ensure the training set and the validation set reflect the properties of the phenomenon to the same extent.

### 3.5. Validation of individual outputs.

As shown above, the validation set - if used with due precautions - can be used to assess how well a particular weight state has learned some features of the trained data. This is useful in comparing different weight states obtained during the training, perhaps with different training mechanisms or even different architectures. This approach could help select a particular architecture/weight state for a given training set.

In this section, a different idea of validation will be discussed. Let us suppose the training has been performed and a particular weight state has been obtained. This weight state is now used to process new data for which the correct output is not known. New input is presented to the net and some output is obtained. Can this output be assessed? Can one evaluate the degree of confidence one can have in this output?

Bishop in [Bishop, 1993] presents a method for validating individual outputs obtained from the net during its normal use and applies it to networks trained by minimising a sum of squares error function. The idea is that the network will perform well in those regions of input space for which there is enough data in the training set and badly in those regions for which the data is insufficient. The training data can be used to calculate an estimate  $\hat{p}(x)$  of the density of the input data  $p(x)$ . A standard Parzen window approach with gaussian kernel functions is used:

$$\hat{p} = \frac{1}{(2\pi)^{\frac{d}{2}} \sigma^d} \sum_{q=1}^n e^{-\frac{|x-x^q|^2}{2\sigma^2}} \quad (4)$$

In this expression,  $x^q$  represents the q-th data point from the training set containing  $n$  points, and  $d$  is the dimensionality of the input space. When the network is in use, the new data is used to calculate a the value given by the estimate  $\hat{p}(\hat{x})$  for the density of the input data. If this value is very low, the input is in a region of the input space very different from the region from which the training data has been collected and the output of the network can be misleading in the sense that it does not reflect the behaviour of the underlying process.

How different is very different? A threshold could be used to separate the familiar from the unfamiliar data. In order to determine the threshold, two data sets are used. A set is used to train the net and the other one is used to calculate typical values of the estimate  $\hat{p}(\hat{x})$  for data which is to be regarded as new.

The crucial difference between the validation used in optimising the topology/weight state and the one used in validating individual outputs is the choice of the training and validation sets. In the first case, when the validation set is used to optimise the network, the validation data must be similar to the data the network will work with (in generalisation). In the second case, when the validation set is used to determine the threshold between familiar and unfamiliar, the data in the validation set must be dissimilar to the data actually used in training. This is precisely because this validation set is supposed to give an idea about what the network's output for unfamiliar data looks like. To illustrate the idea, let us consider the example of a network used in character recognition. For such a net, the validation set used in optimising the topology will contain instances of different characters which are different from those used to change the weights but which are valid characters. This will give an indication about how the net will perform after training. Eventually, these instances can be included in the training set. The validation set used to calculate the density threshold, will contain noise, misleading instances of characters and everything else we want the net to signal as unknown. The instances in this validation set are not valid and will not be included in the training set at any stage.

The main disadvantages of this technique are that i) an estimate of the input density must be calculated for each input and ii) the assessment depends very much on the choice of the data used to calculate the threshold.

Courrieu in [Courrieu, 1994] considers the same problem of validating individual outputs and gives three algorithms for estimating the domain of validity of feedforward neural networks. He points out that "an educated system should be able to assess its own ability to treat a given problem, and potentially, to situate that problem in one domain or another if the system is competent in several domains". The idea is the same as in [Bishop, 1993]: to compare the position of the generalisation instance (the new input) with respect to the position of the training instances. It is shown that the generalisation ability of a neural network is much better if the generalisation instance is inside the convex polytope determined by the patterns in the training set than if the generalisation instance is outside it. The three

algorithms proposed use the convex polytope, a neural approximation of it and a circumscribed sphere respectively to approximate the domain of validity of a network.

A disadvantage of this technique is that there is no reference for the patterns which are outside the region of the training patterns. The technique can offer a qualitative information: the new pattern is outside/inside the convex hull and a quantitative information: the exteriority. The technique should be completed with the threshold evaluation procedure able to calculate how far outside the trained region a pattern can be allowed to be for its generalisation to be considered valid.

In conclusion, these techniques allow an assessment of the degree of confidence of the network's output during its normal use. The main idea is that a network will perform much better in those regions of the input space from where the training patterns have been drawn than in any other regions. This is to say that the network is much better at interpolation than extrapolation.

### **3.6. Generalisation as an approximation.**

Generalisation was defined as being both the property of a neural network to give outputs for untrained inputs and the I/O mapping obtained using this property. We shall concentrate on this latter meaning.

If the existence of an underlying function (as opposed to an underlying stochastic process) is assumed, the generalisation can be seen as its approximation and the training process can be seen as a search for a good (the best if possible) approximation or model. This point of view could help us to identify further problems regarding assessing generalisation.

The approximation problem as usually defined in approximation theory [Poggio, 1990] involves measuring the distance  $\rho$  between the function to be approximated  $f$  and its approximation  $F$ . This distance is usually introduced by a norm.

The approximation problem is: If  $f(X)$  is a continuous function defined on a set  $X$ , and  $F(W,X)$  is an approximating function that depends continuously on  $W \in P$  (where  $P$  is the weight space i.e. a subset of  $R^n$ ) and  $X$ , the approximation problem is to determine the parameters  $W^*$  such that:

$$\rho[F(W^*, X), f(X)] \leq \rho[F(W, X), f(X)]$$

for all  $W$  in set  $P$ . If  $W^*$  exists, it is called the best approximation.

There are three elements involved in the definition of the approximation problem: the function to be approximated (modelled)  $f$ , the approximating (modelling) function  $F$  and the distance  $\rho$  and each of them can influence assessing generalisation.

### 3.6.1 Generalisation with respect to classes of problems.

The distance used in the problem's definition can be more or less appropriate to a particular class of problems.

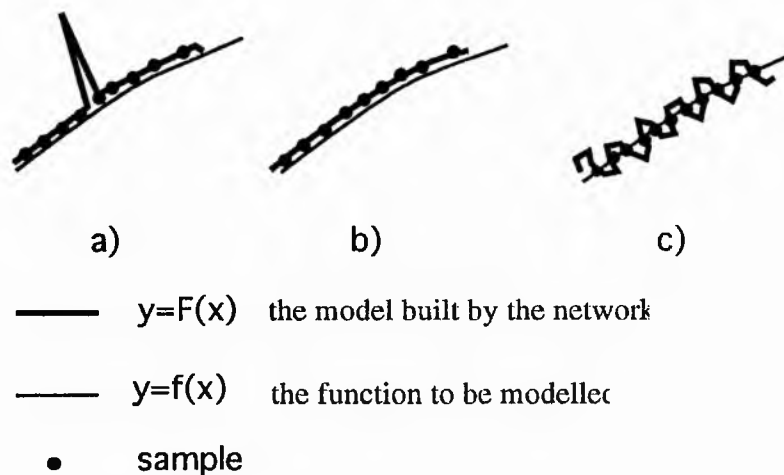


Fig. 9.  $y=F(x)$  is the generalisation (the model found by the net),  $y=f(x)$  is the underlying function and is the same for all a, b and c. In different situations three, two or only one of them is an acceptable generalisation.

Let us consider the example in fig. 9 in which  $y=F(x)$  is the generalisation to be assessed and  $y=f(x)$  is the underlying function. If one is interested in the area under the function, all three generalisations are equally good because they give the same area. If one is interested in the error for any  $x$  value being below a given error limit, only (b) and (c) will be good. Finally, if one is interested in the path's length, only (b) is acceptable. Essentially, this reduces to the definition of the distance between  $f$  and  $F$ .

The conclusion is that through its dependence on the definition of the distance used, the generalisation assessment depends on the use of the generalisation results i.e. the class of problems. In this context, a class of problems is defined (see paragraph 3.2.1) as triplet of an I/O pattern set, an error measure and an error limit.



### 3.6.2 Generalisation with respect to types of underlying functions.

Can one say that radial basis functions offer better generalisation properties than the multilayer perceptron? In terms of the approximation problem, can one say that some  $F$  give a better approximation than others?

Let us consider two simple approximation methods:

the Fourier series:

$$f(x) = \frac{a_0}{2} + \sum_{k=1}^{\infty} [a_k \cos(kx) + b_k \sin(kx)]$$

and the Taylor series:

$$f(x) = \sum_{k=0}^{\infty} \frac{f^{(k)}(a)}{k!} (x-a)^k$$

We shall assume that the function  $f$  is such that both series converge (around a point  $x_0$ ) and only a finite number of terms  $n$  from both series can be considered as the desired approximation.

There are functions  $f$  such as polynomials of degree less than  $n$ , for which the approximation given by the Taylor series yields zero error. For the same functions, the approximation given by the Fourier series can give a large error especially if  $n$  is small. On the other hand, there are functions  $f$  such as trigonometric functions for which the situation is reversed: the Fourier series approximation gives zero error and the Taylor series one can be very poor.

For these reasons, one cannot say that one particular type of net (or training algorithm, or architecture) gives a better generalisation than another. One could only say that a particular type of net is better suited than others to modelling certain classes of phenomena. This point of view is sustained by the existence of some functions which cannot be modelled by any neural network. For instance, one could consider the function associating names with phone numbers in a phone book. No matter how many instances one has in the training set, no neural net will be able to give meaningful generalisation. No network will be able to give the correct phone number of a person whose name is fed to the input of the net if the instance was not in the training set [Poggio, 1990].

The generalisation abilities of a network or training algorithm are only modelling abilities and they must be evaluated only with respect to a precise and specified class of problems. In a "Fourier world" in which all the functions are linear combinations of a finite number of sin and cos of different frequencies, a net which tries to fit only these types of functions through the training points would give the best possible generalisations. Such a net would have found the original transfer function in fig. 4. from the samples in fig. 2. On the other hand, such a net would perform rather poorly in a "linear world" in which all the functions are combinations of a finite numbers of linear functions. Any phenomenon with a linear transfer function would be modelled by a function which oscillates badly between the sample points and the net would need a large number of training points to model (poorly) a straight line. Reciprocally, a net which uses piecewise linear interpolation would perform very well in the "linear world" and very badly in the Fourier world. In this context poor performance means that the net would have the tendency to choose generalisations which are poor models for the sampled phenomenon.

In the real world, the functions can be seen as a combination of an infinite (if necessary) number of sin and cos or other basis functions. The real world functions can be always modelled through piecewise linear interpolation, as well. Depending on the particular problem, a Fourier net could perform better or worse than a net using piecewise linear interpolation.

There is one more question to be asked here. Does a multilayer perceptron with  $n$  hidden units generalise better or worse than another one with  $m > n$  hidden units? What is the relation between the number of parameters of a network and its generalisation properties?

One can see the training of a network as a curve fitting problem. Some classical phenomena can immediately be interpreted in the neural network context. Too many hidden units can be seen as too many degree of freedom which can lead to overfitting. Not enough hidden units could lead to underfitting and so on.

In conclusion, the isolated fact that an architecture is richer or poorer than another (using the same mechanism), cannot justify better or worse expectations for generalisation.

### 3.7. Conclusions.

1. Any function passing through the training points is a valid generalisation. If no other information is available, none of these valid generalisations can be assessed as being better or worse than any other.

2. The assessment of generalisation must take into consideration the class of problem in which the generalisation will be used. This class of problems can be considered explicitly or implicitly through the definition of the distance used in comparing different generalisation.

3. A common assessment of the generalisation properties of a net is performed with the following assumptions about the underlying function (2):

- it is as linear as possible
- it is smooth (continuous, differentiable, with continuous derivatives up to a certain order)

4. The number of samples (training patterns) necessary for good modelling of a given function depends on:

- the purpose of the model
- the type of net which builds the model

A training set cannot be said to contain an insufficient (or sufficient) number of patterns if the purpose of the model and the type of net are not known.

Conversely, if the purpose of the model and the type of net are known, the size of the training set can potentially be assessed.

5. Different types of neural network have different generalisation abilities with respect to a given class of problems. One cannot say that a certain type of net yields a better generalisation than another if the class of problems is not specified. On the other hand, if the class of problems is known, one can select the net with the most suited generalisation properties.

6. Different classes of problems can be more or less easily modelled by a given type of neural network. There are problems (random associations) which do not allow generalisation at all, for any type of network. At the opposite extreme there are the

functions which use mechanisms similar to the mechanism used by the net (linear functions for linear nets, gaussians for radial basis functions, etc.).

7. If the amount of available data is small, the cross-validation is the most useful validation technique because it offers a reliable estimate of the future performance and allows all the patterns to be used in the actual construction of the network. If the amount of data is large or very large, the test sample estimation offers a reliable estimation of the future performance with less effort than the cross-validation. However, measures must be taken to ensure that the data in the validation set is representative.

8. In general, the generalisation of a network is quite reliable if the generalisation needs interpolation and far less reliable if extrapolation is needed. The position of a new pattern with respect to the training set can be used as an indicator of the reliability of the generalisation for that particular pattern.

# **CHAPTER 4**

## **The Constraint Based Decomposition Training Architecture**

### **4.1. Introduction**

This chapter presents a new approach to reach a goal by reaching intermediate sub-goals.

In §4.1, two different possibilities for splitting a task into sub-tasks are presented: the time base decomposition and the constraint based decomposition. An example is used to illustrate the differences between these approaches. Section §4.2 introduces a formal framework for the constraint based decomposition. Terms such as constraint, task, solution, state and discrete solution path are defined. Two types of search are discussed: directed and restricted by subgoals and the connection between the constraint space and the weight space is emphasised in each case.

The sections §4.4 and §4.5 study further the two approaches. In section §4.5, a new constructive algorithm based on the constraint based decomposition and the search restricted by subgoals is presented.

In section §4.6, several experiments with both the search directed and the search restricted by subgoals are presented. These experimental examples are used both to explain the concepts and to illustrate applications of the approaches presented. Other comparative experiments between the new approaches and other known approaches are presented.

Section §4.7 presents the conclusions of the experiments, a summary of the characteristics of the techniques presented and a comparison between the new techniques and other existing techniques.

### **4.2. Time based decomposition (TBD) vs. constraint based decomposition (CBD).**

#### **4.2.1 An example.**

A possible approach to solving a problem is "divide and conquer". The task is split into many simpler tasks which are solved individually. Two fundamentally different

methods for splitting a complex goal into subgoals will be discussed in the following: time based decomposition and constraint based decomposition.

Let us consider a robot with a humanoid anatomy situated in the middle of a room with the task to open the door. Such a task is complex. One feature is that the robot has to move towards the door. Perhaps at the same time, it will move its arm, raising it from its normal position along the body towards the level of the door knob. Concurrently, it will move its fingers preparing them to grasp the door knob. During this complex movement, the head and the eyes must move in such a way that the door knob is kept in the centre of the visual field independently of the position of the body.

Let us suppose we ask a human to perform the task. We are going to record their solution which is one of the many possible solutions and use it to teach our robot. The solution will be a sequence of intermediate positions, a path  $P$  in the space  $S$  of all the possible positions. We call this path a solution path. Now, we could sample this path by choosing a number of intermediate positions:  $(p_1, p_2, \dots, p_n)$ . This is a discrete solution path. The first subgoal of the system is to reach the first point on the path, the second is to reach the next point and so on. Any complex task for which we know (or could design) a path can now be learned. This type of decomposition will be called time based decomposition.

There exists, however, another possibility to split the task into sub-tasks. For the robot to accomplish the task, a set of constraints must be satisfied e.g. the robot must be near the door (i.e. the distance between the robot's mass centre and the door knob must be less than the arm's length), the hand must be at the height of the door knob, the fingers must be open so that grasping the knob is possible, etc. The task is characterised by a set of constraints  $(r_1, r_2, \dots, r_p)$ . This set of constraints is independent of the intermediate states the subject used to reach the final state.

One could consider a constraint space with one dimension for each constraint in the constraint set. Fig. 1 shows a possible training path for a time based decomposition. The variables characterising the constraints vary all at the same time. In each step, each of them will come closer to the value which characterises the solution.

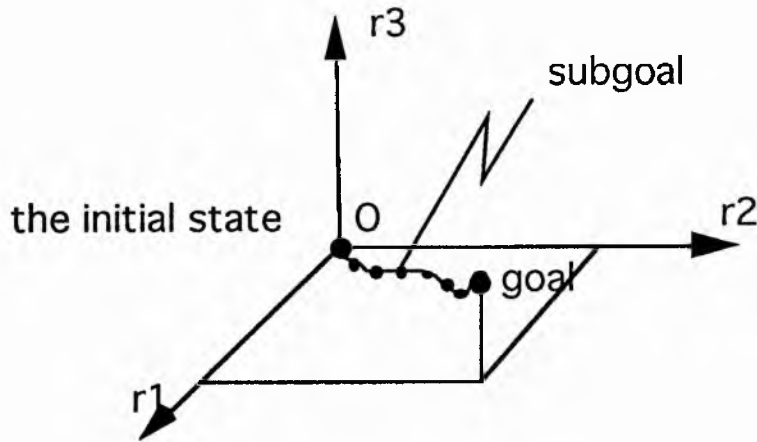


Fig. 1. A time based decomposition (in constraint space). The network is trained with intermediate targets. Each intermediate target (sub-goal) is characterised by the same number of constraint values as the original training set.

Fig. 2 shows a training path for a constraint based decomposition. The subgoals are defined such that the first one (G1) includes the first constraint, the second one (G2) the first two constraints and so on. The first step of the training takes the net into the subspace ss1 corresponding to the correct value of the first constraint. The search for the solution of the second subgoal will be performed in the subspace ss1 which is a subspace with  $n-1$  dimensions of the  $n$  dimensional constraint space. The search for the solution of the next subgoal will be performed in a subspace ss12 with  $n-2$  dimensions (ss1 intersected with ss2) and so on.

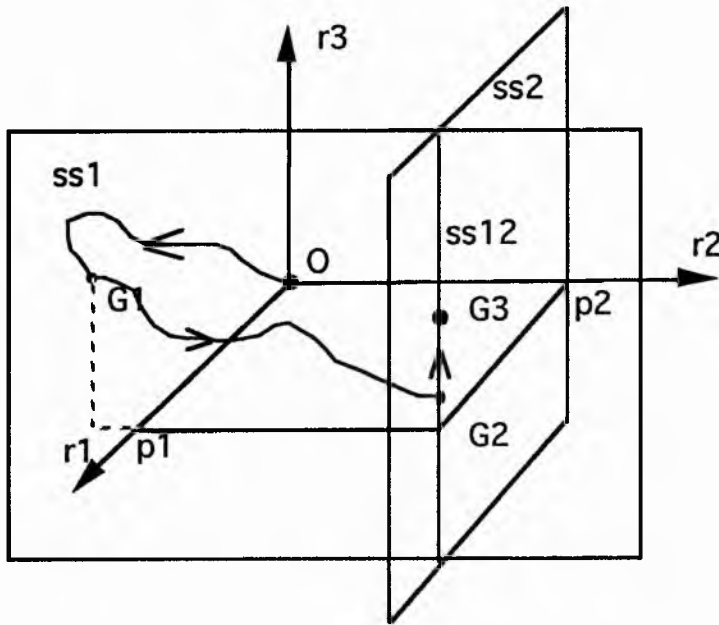


Fig. 2. A constraint based decomposition (in constraint space). Each subgoal asks the satisfaction of one constraint more than the previous subgoal. The search for a solution of a subgoal is performed in a sub-space of the constraint space.

### 4.3. Theoretical framework

The notions of constraint based decomposition and time based decomposition will be now refined with a view to using them in a neural network context. For this reason, the definitions that follow will use terms specific to the given framework of training a neural network. However, the ideas are general and the definitions could be modified for a more general framework.

#### 4.3.1 Definitions

**Definition.** A **constraint** is a condition necessary but not sufficient for the solution. It must be possible for the solution to be expressed as a set of non-contradictory constraints. A constraint can be associated with a constraint variable whose values show the extent to which the constraint is satisfied.

**Definition.** A **task** is defined by a set of constraints  $(r_1, r_2, \dots, r_m)$ . This set of constraints defines a point  $p$  in a constraint space. The **solution** of a task is a point  $W$  in weight space which satisfies the given set of constraints.

**Definition.** Given a task defined by the set of constraints  $(r_1, r_2, \dots, r_m)$ , a **constraint state**  $p$  is an ordered set of constraint values  $(vr_1, vr_2, \dots, vr_m)$ .



**Definition.** A discrete solution path  $P$  is an ordered set of constraint states  $P=(p_1, p_2, \dots, p_n)$ .

**Observation:** In a constraint based decomposition the number of constraint variables defining a subgoal varies but if a variable is present in the definition of a subgoal then this variable must contain the value characterising the solution. In a time based decomposition, the number of constraint variables remains constant and equal to the number of constraints of the problem but their values vary at each stage (for each subgoal).

**Definition.** Given a task defined by the set of constraint values  $(vr_1, vr_2, \dots, vr_m)$ , a **time based decomposition (TBD)** is a discrete solution path  $P=(p_1, p_2, \dots, p_n)$  with the property that  $p_1$  is the initial point,  $p_n$  is the solution and each state  $p_i$  satisfies a set of intermediate constraint values  $(vr_1^i, vr_2^i, \dots, vr_m^i)$ .

In this definition,  $m$  is the number of constraint values defining the solution (the dimensionality of the constraint space),  $n$  is the length of the discrete solution path (the number of intermediate subgoals (including the final one) and  $i$  indexes the subgoals.

**Definition.** Given a task defined by the set of constraint values  $(vr_1, vr_2, \dots, vr_m)$ , a **constraint based decomposition (CBD)** is a discrete solution path  $P=(p_1, p_2, \dots, p_n)$  with the property that  $p_1$  is the initial point,  $p_n$  is the solution and each state  $p_i$  satisfies the constraint values  $(vr_1, vr_2, \dots, vr_i)$ .

Once more,  $m$  is the number of variables characterising the solution,  $n$  is the length of the discrete solution path and  $i$  indexes the subgoals.

#### **4.3.2 Constraint implementation. CBD as a method to perform a dimensionality reduction in the weight space.**

The above definition of a constraint is very general. For it to be used in training a neural network, a constraint should be defined in terms of input/output patterns. This would establish the connection between the theoretical framework of the constraint based decomposition and the practical problem of training a given input/output set.

For these purposes, a constraint is defined as obtaining the correct output for the patterns in the training set which are situated in a given region of the input space or

equivalently the construction of the desired I/O surface above a given region of the input space. When the output is the correct one, the constraint is satisfied.

This definition satisfies the conditions for a constraint because:

The whole I/O surface can be cut into pieces corresponding to disjoint regions of input space  $\{a_1, a_2, \dots, a_3\}$ . In order for the I/O surface  $S$  to be the goal I/O surface  $S_g$ ,  $S$  must be equal to  $S_g$  in all of the regions  $a_1, a_2, \dots, a_3$ . Therefore,  $S = S_g|_{a_i}$  (the condition that  $S$  be equal to  $S_g$  in the limited area  $a_i$ ) is a necessary but not sufficient condition.

The solution, i.e. the goal I/O surface, can be expressed as a set of non-contradictory constraints:  $S$  is a solution if and only if  $S = S_g|_{a_i}$  for any  $i$  from 1 to  $n$ , where  $n$  is the number of areas the input space has been cut into. Due to the fact that the  $a_i$  are disjoint by definition, the constraints cannot be contradictory.

The CBD training starts by training the first subgoal which requires the satisfaction of the first constraint. The second subgoal will ask for the satisfaction of the first two constraints. Therefore, the search for the solution of the second subgoal is performed in a subspace with  $n-1$  dimensions of the  $n$  dimensional constraint space.

The interesting case is when the reduction of dimensionality in the constraint space can be put into correspondence with a reduction of dimensionality in the weight space. In this case, the weights found in one subgoal training will be preserved unchanged and will be a part of the final solution. Having as few dimensions in the weight space as possible is one of the characteristics of the ideal training situation.

The shape and the size of the regions of input space used in defining the constraints is very important. The specific shape and the size chosen should depend on the problem. Ideally this should be done automatically, by the training algorithm.

#### **4.3.3 Search directed by subgoals**

The technique characterised as search directed by subgoals provides a situation in which there is only a weak coupling between the constraint space and the weight space. A reduction of dimensions in the constraint space may, but does not necessarily, correspond to a reduction of dimensions in the weight space.

#### **4.3.4 Search restricted by subgoals.**

In another technique, called search restricted by subgoals, there will be a strong connection between the constraint space and the weight space. A reduction of dimensions in the constraint space is put into a direct correspondence with a reduction of dimensions in the weight space. The weights found by training a subgoal will become a part of the final solution. Here, the Constraint Based Decomposition net tries to implement the idea of a search restricted by subgoals.

#### **4.4. Search directed by subgoals**

The simplest form in which the CBD idea can be implemented is to define the subgoals by splitting the training set into subsets formed by nearby patterns. This is roughly equivalent to splitting the input space into disjoint regions and taking as a training set of a subgoal, the patterns in this region. A constraint is getting the correct output for a subset of the training set. A subgoal is the training of an increasing number of constraints.

In order to check the effects of this CBD, one could simply train with a standard weight change algorithm such as backpropagation with momentum (see sections 2.2.2 and 2.3.1 in chapter 2) the subgoals corresponding to the chosen constraints. In constraint space, the net is asked to reach the first subgoal. From this point, the net is trained with the second subgoal. No measures are taken to ensure that the net will remain in the subspace corresponding to the first subgoal. The question is the extent to which the net will be able to preserve the information obtained during the training of the first subgoal in the training of the second one.

The answer to this question will be given by the evolution of the error for the patterns in a subgoal subset during the training of the subsequent subgoal. If this error remains small, it will mean that the net's outputs to the patterns in the first subset are still correct. In other words the trajectory of the training in constraint space remains close to the subspace determined by the first subgoal. The search for the solution to the subsequent subgoal is directed by the subspace corresponding to the preceding one. If the error goes up, it will mean that the first few weight changes in each training session throw the net far from the subspace corresponding to the precedent subgoal. In this case, all subgoal training sessions bar the last one are wasted.

Such an experiment can also emphasise the importance of the pattern presentation algorithm as a part of the training algorithm. If the result of the training can be substantially changed by changing only the pattern presentation algorithm, a training algorithm must be seen as the combination of a weight changing algorithm and a pattern presentation algorithm rather than a weight changing algorithm alone. A substantial change would be for instance the success of the CBD pattern presentation algorithm in some problem where the batch pattern presentation algorithm fails. Both should use the same weight updating rule.

The CBD pattern presentation algorithm tries to ensure that, for each training, the position of the initial weight state in relation to the position of each subgoal is good. The network is asked to learn a little at a time. This is achieved by training exclusively on pattern sets containing mostly patterns that the net is already able to respond to correctly.

#### **4.5 Search restricted by subgoals. The Constraint Based Decomposition net.**

##### **4.5.1 The description of the CBD algorithm.**

Since the purpose is to train only a few weights at a time and to keep those weights unchanged afterwards, the idea of constructing the net during the training comes naturally in one's mind. The CBD algorithm is formed by a CBD pattern presentation algorithm, a construction mechanism for building the net and a weight change algorithm for a single layer network (perceptron training for instance). Since the network is constructed during the training and its architecture is intrinsically connected with the training, the resulting network architecture and weight state will be called the constraint based decomposition net (the CBD net). However, the constraint based decomposition approach is not limited to this particular implementation. The pattern presentation algorithm presented in section 3 above falls into the same broad constraint based decomposition approach and perhaps there are many other possible implementations of the same idea.

##### **4.5.1.1. Two classes.**

To illustrate the CBD algorithm, let us consider the example of a classification problem. In the first instance, we shall consider only two classes C1 and C2 in an  $n$  dimensional input space. The problem is defined by a set of patterns for each class. There are two output units O1 and O2, one for each class.

The CBD algorithm starts with the input units, one unit (which will become a hidden unit eventually) and the bias unit (permanently set to 1). For classification problems one can use threshold units with  $(-1,+1)$  output range and 0 threshold. It is worth mentioning that the limitation to binary units is not imposed by the algorithm but has been chosen here for presentation reasons. The results can be easily extended to a network able to train analogue targets but this extension will not be presented here<sup>1</sup>.

Let  $C_1 = \{x_1^{C1}, x_2^{C1}, \dots, x_m^{C1}\}$  be the set of patterns in the class  $C1$  and  $C_2 = \{x_1^{C2}, x_2^{C2}, \dots, x_n^{C2}\}$  be the set of patterns in class  $C2$ .

The first stage is to construct a hidden layer (the hyperplane layer) which has a hidden unit for each hyperplane necessary for the separation of the regions belonging to different classes. The result of this stage is a set of hyperplanes  $h_1, h_2, \dots, h_k$  and a set of regions separated by piece wise linear boundaries. Each of these regions, can be described as a **term**  $T_i$  of the form  $T_i = (\text{sign}(h_1)h_1 \dots \text{sign}(h_k)h_k, C_j)$  where  $\text{sign}(h_i)$  can be 1,-1 or nil and  $C_j$  is the class to which the region belongs.

Each hyperplane divides the space into two regions (half-spaces) one positive and one negative. A hyperplane and its sign form a **factor**. A factor is used to represent one of the two half-spaces determined by the hyperplane. A term is obtained by performing a logical **and** between factors. Not all the hyperplanes must contribute with a factor to all terms. Finally, a logical **or** is performed between terms in order to obtain the expression of the solution for each class.

---

<sup>1</sup> A network for analogue targets could use the same first layer of hidden units as the network for classification. Since the algorithm ensures that the hyperplanes are positioned in such a way that homogenous regions (containing only patterns from the same class) are formed, a subsequent layer can be trained for the desired analogue values without worrying about linear separability. In other words, once the first hidden layer has been designed (weights included), the problem is linearly separable in hidden unit space and can be solved by the next layer.

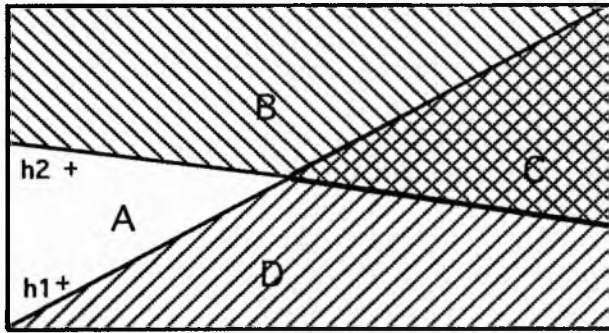


Fig. 3 An illustration of the formalism used. For instance, region A can be described by a term  $T_1 = h_1 h_2$  (read  $h_1$  plus and  $h_2$  plus).

An example of the formalism used is given in fig. 3. Region A can be described by a term  $T_1 = h_1 h_2$  (read  $h_1$  plus and  $h_2$  plus), region B by  $T_2 = h_1 \bar{h}_2$  ( $h_1$  plus and  $h_2$  minus), C by  $T_3 = \bar{h}_1 h_2$  and D by  $T_4 = \bar{h}_1 \bar{h}_2$ . If the solution was the union of regions A and C, it could be expressed as  $S = h_1 h_2 + \bar{h}_1 h_2$  i.e. ( $h_1$  plus) and ( $h_2$  plus) or ( $h_1$  minus) and ( $h_2$  minus) (note the usual priority of and/or operators).

The final network will have three layers. The first layer will implement the hyperplanes necessary for the construction of the piece wise linear boundary, the second layer will implement the logical and necessary to construct the terms and the third layer (the output layer) will perform a logical or between different terms. In general, the last layer will have a unit for each class.

In the particular case of only two classes, since a pattern belongs to only one class, one of the output units can be omitted. The network can have only one output unit which is on for patterns from one class and off for patterns from the other class. In this case, the union of the areas of input space associated with each class will be the whole input space. If 'don't know' regions are necessary or if there are more than two classes, an output unit will be used for each class.

The algorithm is presented as a recursive procedure. Its parameters are: a region of the space (initial value = whole space), the training set divided into two sets, one for each class (initial value = the whole training set) and a factor (initial value = nil). The factor describes the region and nil corresponds to the whole space.

The CBD algorithm starts by building a subgoal with only two patterns, one from each class. A unit (which is to be a hidden unit in the final net) will be added and trained until it separates the two patterns. As discussed in the chapter on training, this training problem is the simplest problem one can have: only one layer and only

one unit. It is assumed that this training will succeed. The assumption refers to the ability of the algorithm to find a solution because a solution will always exist and it is easy to find (it can actually be calculated from the patterns itself). Another possibility is to start with  $n$  patterns where  $n$  is the dimension of the input space. This problem is always linearly separable if the patterns are in general positions i.e. if they are linearly independent (in input space). Although the probability of obtaining  $n$  linearly dependent input patterns by randomly choosing  $n$  patterns in an  $n$  dimensional input space is zero in practice, the method of starting with only two patterns is more elegant because the separability is guaranteed.

Let  $h$  be the hyperplane obtained by this training. This hyperplane will be saved. A new pattern (from any class) will now be added to the current subgoal. The same unit will be trained again. The training problem is again the simplest possible: one layer, one unit and the pattern set contains a single misclassified example. If the training succeeds, the latest pattern will remain in the current subgoal and the new hyperplane will be used subsequently. If the training fails, the old hyperplane will be restored and the pattern will be deleted from the current subgoal. Because the training has failed, it is assumed that the correct classification of this pattern is not possible with the current hyperplane such as all other patterns previously correctly classified remain so. The algorithm postpones the correct classification of this pattern to a later stage when this correct classification will be achieved with a different hyperplane.

The process of trying to improve the position of the current hyperplane by considering more patterns, one at a time, continues until all the patterns in the training set have been considered. At all times, the pattern set poses the simplest possible problem: at most one misclassified pattern.

A termination problem arises here: when to stop the training of a particular subgoal. If the training is successful, the termination condition is clear: stop the training when all the patterns in the current subgoal are correctly classified (or when the error is below the error limit). If the training is not successful, the simplest termination condition which can be applied is a time-out condition. Thus, the failure of a training can be detected by imposing a time-out condition in number of epochs or monitoring the weight changes and stopping the training when the error evolution becomes asymptotic. More sophisticated termination criteria will be discussed in chapter 5.

The hyperplane resulted at the end of this process will divide the space into two half-spaces  $h_+$  and  $h_-$ . Now consider the whole original training set again. If  $h_+$  contains only patterns in the same class  $C_j$ , this half-space can be labelled as belonging to class  $C_j$ . If  $h_-$  contains only patterns in the same class, it will be labelled as well. If at least one of the half-spaces is not homogeneous (i.e. it contains patterns from more than one class), the algorithm will be called recursively in that particular half-space.

In general, the algorithm is called to separate the patterns in a region obtained as an intersection of half-spaces and the current pattern set contains only those patterns from the original pattern set which are in the current region. This region is characterised by a term which contains all the hyperplanes (with their correspondent signs) which define the given region.

Let us assume the algorithm is currently working in a region described by a term, be it `old_term`. Let us assume that the hyperplane  $h$  was positioned during the current call of the algorithm.  $h_+$  is now checked for consistency. If it contains patterns from one class ( $C_j$  for instance),  $h_+$  will be added to the current factor and the result classified as class  $C_j$ . The resulting region will be the intersection between the current region and  $h_+$ . Therefore, the new term characterising the new region will be (`old_term and  $h_+$` ). If  $h_+$  is not homogeneous (it contains patterns in both classes) the algorithm will be applied again to the new region. The same is done for  $h_-$ .

This algorithm regards the satisfaction of a constraint as the construction of a homogeneous region i.e. a region containing only patterns from the same class. Once a homogeneous region has been found, a constraint has been satisfied i.e. the I/O surface implemented by the net is the desired one above a given region of the input space. This region will not be affected by the subsequent training and will become a part of the final I/O surface.

This idea of building the desired I/O surface from elements which satisfy the requirements of the problem locally is not new. The same idea is exploited by radial basis functions (RBF's) for instance (see [Broomhead, 1988], [Moody, 1989], [Musavi, 1992], [Poggio, 1990]). However, in the case of a surface constructed from radial basis functions, the basic 'building blocks' used are always the same, imposed by the chosen radial basis function. In CBD's case, the building blocks are homogeneous areas found by the algorithm during the training. These areas can have



any shape (with piece wise linear boundaries) and therefore this approach can be more flexible than the RBF's unique building block approach.

Furthermore, the CBD algorithm presented here uses a novel pattern presentation algorithm based on the constraint based decomposition approach. Once a homogeneous region is found, the patterns contained in this region will be removed from further consideration. This is the implementation of the dimensionality reduction in constraint space characteristic of the constraint based decomposition approach. At the same time, once a homogeneous region is found, the weights defining the hyperplanes bounding this region will be stored and will become a part of the final solution. This is the implementation of the strong connection between the constraint space and weight space characteristic of the search restricted by subgoals approach.

The algorithm for the first stage (building and training the hyperplane layer) is presented in fig. 4.

The next stage is very simple and does not need training at all. CBD builds another layer with a unit for each term  $T_i = \text{sign}(h_1)h_1 \dots \text{sign}(h_k)h_k$ . These layers are similar to those used by the entropy nets (see [Sethi, 1990a], [Sethi, 1990b]). However, Sethi's hyperplanes come from the a priori construction of a tree classifier using a standard design technique. Furthermore, in [Sethi, 1990a], the use of a single feature decision function at every non-terminal node means that the hyperplanes can only be perpendicular to the axes. In this paper, Sethi suggests the use of the sigmoidal activation function to implement boundaries which are not perpendicular to the feature space's axes. In [Sethi, 1990b], the use of hyperplanes at different orientations is suggested but the positioning of these hyperplanes is left entirely to tree design procedures like AMIG and CART [Breiman, 1984].

It is well known in the literature that a threshold neuron can implement a logical function such as logical AND or logical OR. However, a specific algorithm for setting the weights in the AND and OR layers is given in the following.

Let us consider the unit associated with  $T_i$ . The bias weight  $w_{\text{bias}}$ , will be set at an arbitrary negative value (e.g. -0.5). Since the unit implements the term  $T_i$ , the unit will be connected only with the units on the first hidden layer corresponding to the hyperplanes in  $T_i$ .

The sign of each weight will be given by the sign of the hyperplane in the term  $T_i$ . The absolute value of the weights depends on the fan-in (the number of neurons on the previous layer the current unit is connected to) and can be calculated in the following way. Firstly, the unit must be off (i.e. its excitation must be below the threshold) even if only one of the neurons on the previous layer has the wrong output i.e. classifies incorrectly the given input pattern. That is:

$$x \cdot (fan\_in - 1) - x < threshold \quad (1)$$

The first term is the weight  $x$  multiplied with the number of units which have the correct output (all bar one which is wrong and whose output will be -1). The second term takes into account the effect of the wrong neuron whose output will be -1.

Secondly, the neuron must be on (i.e. its excitation must be above the threshold) when all neurons (corresponding to hyperplanes in  $T_i$ ) have the right excitation. That is:

$$x \cdot fan\_in > threshold \quad (2)$$

Therefore, the absolute values of the weights will be all equal to  $x$  where  $x$  is any value in the solution interval of the following inequalities:

$$\begin{cases} x > \frac{threshold}{fan\_in} \\ x < \frac{threshold}{fan\_in - 2} \end{cases}$$

where  $fan\_in$  is the number of hyperplanes present in  $T_i$ .

The first inequality ensures that the unit will be turned on if all of the units are in the state required by the sign of their corresponding factors. The second inequality ensures that the unit will remain off if even a single unit has the wrong activation. For the chosen type of neurons, the threshold is the absolute value of the bias weight. Such a unit implements a logical **and** and will be turned on if and only if the input pattern is in the region described by the term  $T_i$ .

There will be a unit on the second hidden layer for each term in the solution given by the algorithm. Finally, another layer will implement a logical **or**. This layer (the output layer) will contain a unit for each class (2 units in this case) and each unit

will be connected with the terms corresponding to its class on the previous layer. The weight can have any value greater than the threshold (any value greater than 0.5 in this case).

In conclusion, the CBD algorithm builds a net with 3 layers of active weights. The first layer implements hyperplanes which separates the patterns into regions containing only patterns in the same class. The second layer implements a logical **and** between different hyperplanes. In the set theory language this layer implements an **intersection** between half-spaces given by different hyperplanes. Each unit on this layer will be activated only by input patterns situated in a homogeneous region of the input space and can be associated with their output class. The third layer implements a logical **or** between units on the second layer. In other words it performs the **union** of different regions corresponding to the same class. The type of final architecture of the net is presented in fig. 5.

```

separate ( region, C1=set of patterns in class 1, C2=set of patterns in class 2, factor )
• Build a subgoal S with patterns  $x_1^{C1}$  and  $x_1^{C2}$  taken at random from C1 and C2. Delete
 $x_1^{C1}$  and  $x_1^{C2}$  from C1 and C2.
/* choose a pattern from each class */
• Add a hidden unit and train it to separate  $x_1^{C1}$  and  $x_1^{C2}$ . Let h be the hyperplane which
separates them.
/* separate them */
• For each pattern p in C1 U C2.
/* optimise the position of the separating
hyperplane so that as many patterns as possible
are separated by the same hyperplane */
    • Add p to the current subgoal S
    • Save h in h_copy
    • Train with the current subgoal S
    if not success then
        • Restore h from h_copy
        • Remove p from S
• Let new_factor = factor and (h,'+')
• If the positive half-space determined by new_factor contains only patterns in the same class
Cj then
/* if this region is consistent */
    • Classify new_factor as Cj /* done */
else
/* else call the procedure recursively in the smaller region */
    • Delete from C1 and C2 all the patterns which are not in h+. Store the result in
new_C1 and new_C2.
    • Separate( h+, new_factor, new_C1, new_C2, new_factor )
• Let new_factor = factor and (h,'-')
• If the negative half-space determined by new_factor contains only patterns in the same
class Cj then
/* if this region is consistent */
    • Classify new_factor as Cj /* done */
else
/* else call the procedure recursively in the smaller region */
    • Delete from C1 and C2 all the patterns which are not in h-. Store the result in
new_C1 and new_C2.
    • Separate( h-, new_C1, new_C2, new_factor )

```

Fig. 4 The CBD algorithm for building and training the hyperplane layer.

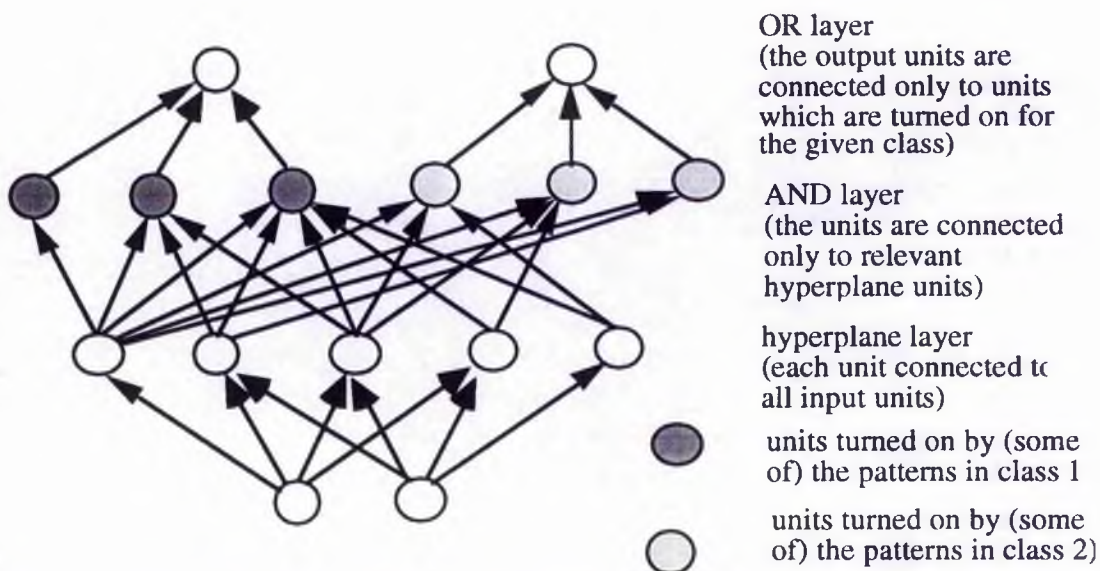


Fig. 5. An example of a complete architecture of a CBD network. This example shows the solution of the problem presented in fig. 18.

Another option is to use only two layers of active weights. The first one which corresponds to the first layer of hidden units is built in the same way and the second one is trained with the delta rule (see chapter 2) or any other well-known learning algorithms for single layer networks. Because the hyperplanes implemented by the units in the first hidden layer are in the correct position, the problem is separable in the hidden layer activation space and a solution exists for the training of the second layer of weights.

#### 4.5.1.2. Proof of convergence for the CBD constructing algorithm (for binary outputs)

A simplified version of the algorithm in fig. 4 will be used to prove the convergence of the CBD training. This simplified version is presented in fig. 6. This simplified algorithm does not store the solution and it does not look for a good solution. An inefficient solution will suffice. In the worst case, the solution given by this algorithm will construct regions containing only a single pattern which is very wasteful.

Let us assume there are  $M$  (distinct) patterns  $(x_i)_{i=1, \dots, M}$  of  $N$  input variables. Each pattern  $x_i$  is associated to a binary output  $y_i = \pm 1$ . The pattern can be seen as belonging to one of two classes  $C1$  and  $C2$ , where  $C1$  contains all the patterns

associated to an output value of +1 and C2 contains all the patterns associated to an output value of -1. It is assumed that the number of patterns  $M$  is finite.

```

main
begin
    region = whole input space
    pattern_set = whole pattern set
    separate( region, pattern_set)
end

separate( region, pattern_set)
begin
    1. if region contains only pattern from the same class then
        endproc
    2. take two patterns, one from each class and remove them from pattern_set
    3. separate these two patterns with a hyperplane hp
    4. region_plus = positive half-space of hp
    5. region_minus = negative half-space of hp
    6. pattern_set_plus = {patterns in pattern_set and in region_plus}
    7. pattern_set_minus = {patterns in pattern_set and in region_minus}
    8. separate( region_plus, pattern_set_plus )
    9. separate( region_minus, pattern_set_minus )
end

```

Fig. 6. A simplified version of the CBD algorithm.

One can show the correctness of this algorithm in the assumption that, for any two given patterns, a hyperplane which separates them can be found in a finite time. This assumption is used in step 3.

From the termination condition 1 it is clear that if the algorithm terminates, this happens because the region contains only patterns from the same class. Because the two patterns chosen in step 2 are separated (using the assumption) in step 3, and because they are removed from pattern\_set, both pattern\_set\_plus and pattern\_set\_minus will contain at least one pattern less than pattern set. This implies that after  $M-1$  recursive steps the procedure separate will be called with a region containing just one pattern. Such a region satisfies the termination condition 1 and the algorithm will terminate. Therefore, the algorithm is guaranteed to terminate

after at most  $M-1$  recursive steps. This worst case situation happens when the consistent regions containing only patterns from the same class all contain just one pattern. In this case, the input space is shattered into  $M$  regions.

One can note that the termination condition and the recursive mechanism is identical for both the simplified CBD algorithm in fig. 6 and the CBD algorithm in fig. 4. The difference between them is that the CBD algorithm in fig. 4 tries to minimise the number of the hyperplane used by trying to optimise their positions with respect to all the patterns in the current training set (the first "for" cycle in the algorithm in fig. 4). This means that the CBD algorithm cannot perform worse than the simplified algorithm in fig. 6. i.e. it is guaranteed to converge in at most  $M-1$  steps for any pattern set containing  $M$  patterns.

#### 4.5.1.3. Classification of more than two classes.

The purpose here is to separate various inputs into classes when inputs from more than two classes are presented. Let us consider that patterns from  $C_1, C_2, \dots, C_n$  are presented to the net. A 3 class problem is presented in fig. 7. Various approaches are possible.

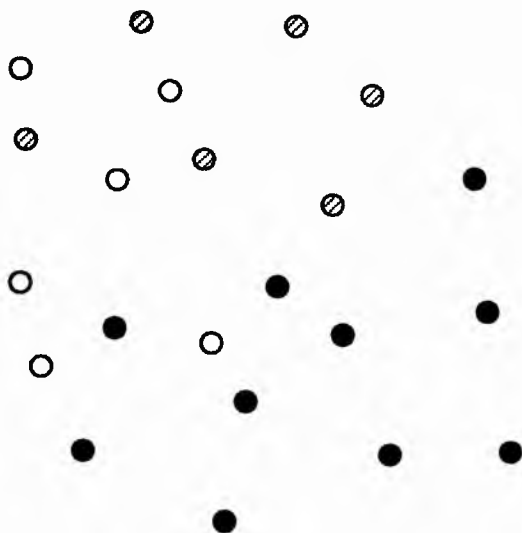


Fig. 7. A 3 class problem in a 2D input space.

One approach is to solve iteratively the multiclass problem as a set of two-class problems. Firstly, the algorithm separates the pattern set into homogeneous subsets  $S_1, S_2, \dots, S_n$ , each of which contains only patterns in the same class. Then, all

pairs of classes are chosen and separated by training a network using the two class version of the CBD algorithm. Let us suppose that classes  $C_1$  and  $C_2$  are chosen to be separated in this first phase. In the example given, the patterns from  $C_3$  are ignored in the first instance and classes  $C_1$  and  $C_2$  are separated. The result of separating  $C_1$  from  $C_2$  (shown in fig. 8) is a set of hyperplanes determining a set of regions  $R_i$ , each region having assigned a class  $C_k$  where  $k$  is 1 or 2.

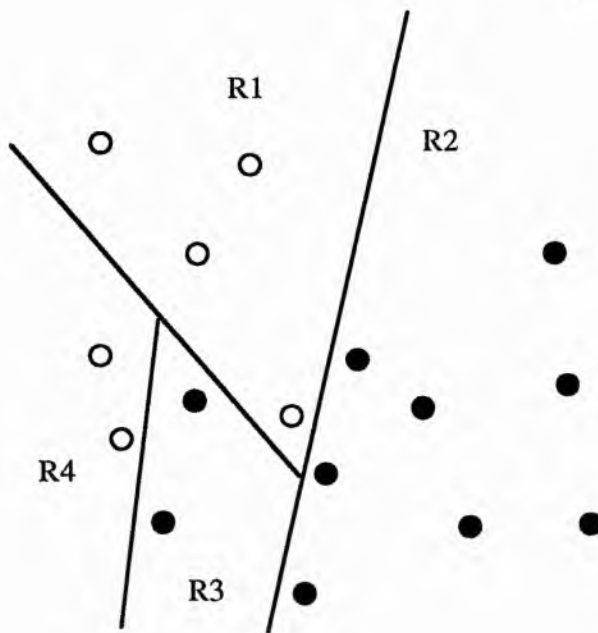


Fig. 8. The first stage in a multiclass separation is solving a 2-class problem. Any two classes are chosen and separated.

Once two classes have been separated, two approaches are possible: iteration on the classes to be separated and iteration on the regions obtained by solving the first two-class problem.

In the first approach, for each remaining class  $C_j$  from  $C_3, \dots, C_n$ , each pattern will be taken and fed to the net in order to identify the region in which this pattern lies. Let this region be  $R_i$  and the class assigned to this region by the first run of the two-class algorithm  $C_{ik}$ . Subsequently, a two-class problem will be solved: separate  $C_{ik}$  from  $C_j$  in  $R_i$ .

In the example, let us consider that a pattern from the third class is randomly chosen. Let this be one of the patterns situated in  $R_1$ . The next problem will be a 2-



class problem: separate the white patterns from the dashed patterns to be solved in R1 as shown in fig. 9

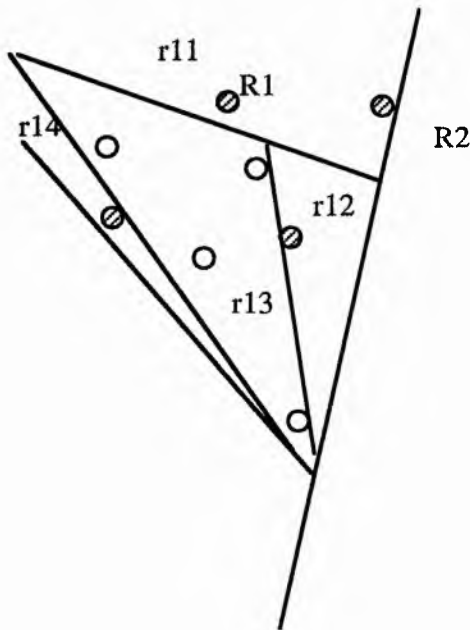


Fig. 9. Iteration on the patterns. In this case one of the patterns of the remaining class (dashed) has been chosen and it happens to be in R1. Consequently, R1 is split so that the classes in R1 (dashed and white) are separated.

Subsequently, all dashed patterns in R1 will be ignored because they are already separated from the patterns in the class to which R1 was initially assigned. Another dashed pattern will be chosen (perhaps in R2) and another 2-class problem will be solved in a limited region of the input space. The process is continued until all patterns in all remaining classes are considered and the problem is completely solved. A possible solution is presented in fig. 10 and the algorithm used is presented in fig. 11.

In conclusion, this approach involves solving a two class problem in the entire input space followed by a iterative check through all the patterns from the remaining classes. For each pattern, if the region in which it lies is not consistent (i.e. contains patterns from more than one class), the separation procedure will be called in that particular region.

In the second case, after two classes have been separated and the space divided into consistent regions  $R_i$ , each region  $R_i$  can be checked for consistency with respect to all classes. If the region is not consistent, the multiclass procedure can be applied recursively. As the regions become smaller at each step and if the training set contains a finite number of patterns, the process will eventually converge to consistent regions which can then be labelled with their correspondent class. Such an algorithm is presented in fig. 12.

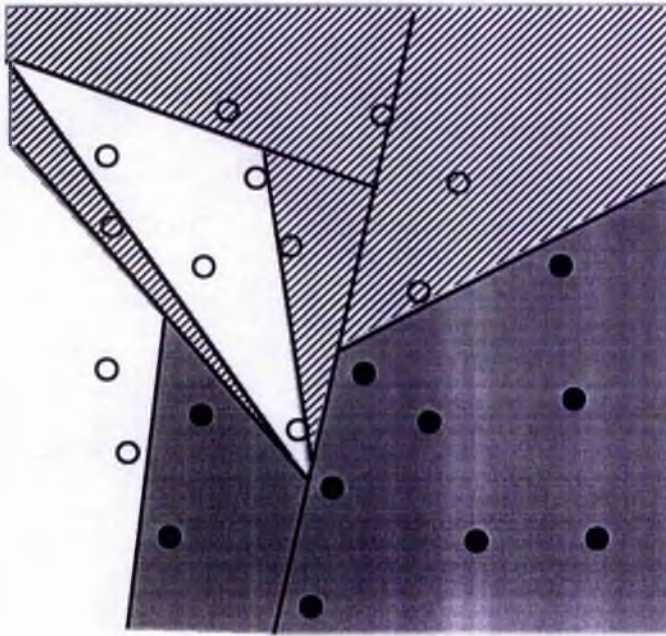


Fig. 10 A solution for the example problem.

**Algorithm 1. (iteration on classes  $C_j$ )**

**separate**  $C_1, C_2, \dots, C_n$ , Region

split the pattern set in consistent subsets (one class in each subset)

build a pattern set with only two classes  $C_1$  and  $C_2$ .

separate  $C_1$  from  $C_2$  in Region (the whole space). The result is a set of hyperplanes determining a set of regions  $R_i$ , each region having assigned a class  $C_{ik}$ .

**for** each class  $C_j$  with  $j$  from 3 to  $n$  **do**

**for** each pattern  $p_{ij}$  in  $C_j$  **do**

        apply  $p_{ij}$  to the input of the net and see in which region is classified.

        let that region be  $R_i$  with its corresponding class  $C_{ik}$ .

        separate  $C_{ik}$  from  $C_j$  in  $R_i$

**end**

Fig. 11 Algorithm 1 for a multiclass classification.

**Algorithm 2. (iteration on regions  $R_j$ )**

**separate**  $C_1, C_2, \dots, C_n$ , Region

split the pattern set in consistent subsets (one class in each subset)

separate  $C_1$  and  $C_2$  in Region

**for** each region  $R_i$  of Region **do**

**if**  $R_{ik}$  is not consistent **then**

        separate whatever classes there are in  $R_{ik}$

**end**

Fig. 12 Algorithm 2 for a multiclass classification.

The problem of multiclass separation can be approached in a different way, which does not involve a first stage of separation of two classes. The previous solutions build a single network provided with output neurons corresponding to each class. When a pattern from a given class is presented to the network, the output neuron corresponding to that particular class is turned on. An alternative approach can build a network for each class. Each net will make the distinction between its own class and any other class. For this purpose, the patterns must be organised in sets corresponding to each class net. Each such set will contain the patterns from one class as the class to be recognised and the patterns from all other classes as the opposite class. Such an algorithm is presented in fig. 13.

**Algorithm 3. (for parallel hardware)**

**separate**  $c_1, c_2, \dots, c_n$  in region is

split the pattern set in consistent subsets (one class in each subset)

**for each class**  $c_j$  **do**

    build subset with the patterns in  $c_j$  as one class and all the other patterns in a different class  $c_{diff}$

**separate\_two\_classes**  $c_j$  and  $c_{diff}$  in region

Fig. 13 Algorithm 3 for a multiclass classification.

**Discussion.**

Algorithm 1 can be very efficient if there is an important difference between the number of patterns in different classes. The classes with the largest number of patterns can then be separated first. If the number of patterns in the remaining classes is not too large, only a few supplementary separations will be needed. In the same case, algorithm 2 can be very inefficient. Suppose there are  $N$  regions after the separation of  $C_1$  from  $C_2$  and just one supplementary class  $C_3$  with just one pattern. In this situation, algorithm 2 takes all  $N$  regions into consideration and their consistency will be checked. Algorithm 1 takes only the pattern in class  $C_3$ , finds the region the pattern is in and separates only in that region. Algorithm 1 eliminates

the useless consistency checks of those regions which are consistent by using the patterns in the remaining classes to identify the regions which are not consistent.

Algorithm 3 separates each class from every other class. Thus, each net will need only the hyperplanes to separate between its own class and the rest, without any concern for the separation between other classes and therefore, each net will be smaller in general, than the net obtained with the any of the previous approaches. On the other hand, the same hyperplane could be useful in the separation of more than one class. In this case, this hyperplane will be implemented by different units in different class nets. Therefore, the total number of neurons used by this approach would in general be greater than the number of neurons used by the previous approaches.

However, algorithm 3 is very convenient in parallel hardware if a piece of hardware can be allocated to each class-net. The convenience comes from the fact that the training can be done in parallel. Each training of one class net is independent of any other training and the parallelism is fully exploited.

#### **4.6. Experiments.**

##### **4.6.1 Search directed by subgoals. Pattern presentation algorithm.**

Two types of experiments were performed. The first type compares the training of a constraint based decomposition approach with respect to the standard training approach (backpropagation with momentum). The second type of experiments investigates in more detail the importance of the pattern presentation algorithm and shows that the search is indeed directed by the subgoals.

##### Experimental details

The experiments were done with a classification network with a 128-20-36 architecture. The net is used to classify alphanumeric characters (10 digits and 26 letters).

The training patterns were obtained from images of car number plates. The image is segmented into number plate and background and the number plate is segmented into characters. Each character area is analysed and a histogram of grey levels is computed. This histogram is assumed to be bimodal. The histogram is analysed to find the two modes. A binarisation threshold is calculated by choosing a grey level in the valley between the two modes.

Since the network has a fixed number of analogue input units (128), a size normalisation problem appears: how to calculate 128 meaningful values from a character area which can have a very different size from one character instance to another. This size normalisation problem is solved in the following way. Each rectangular character area is binarised using the variable threshold (calculated from the histogram) and divided into  $8 \times 16 = 128$  smaller rectangles (Fig. 13b). Because the zone of the image containing a character has a variable size which depends on the particular image, each small rectangle will contain a different number of binarised pixels each time. Using this variable number of binary values, a mean luminance value is calculated for each small rectangle. These 128 luminance values are normalised to a value between 0 and 1 and the result of this normalisation constitutes the input to the net. The output is a vector of 36 elements with all elements zero except the one corresponding to the character presented.

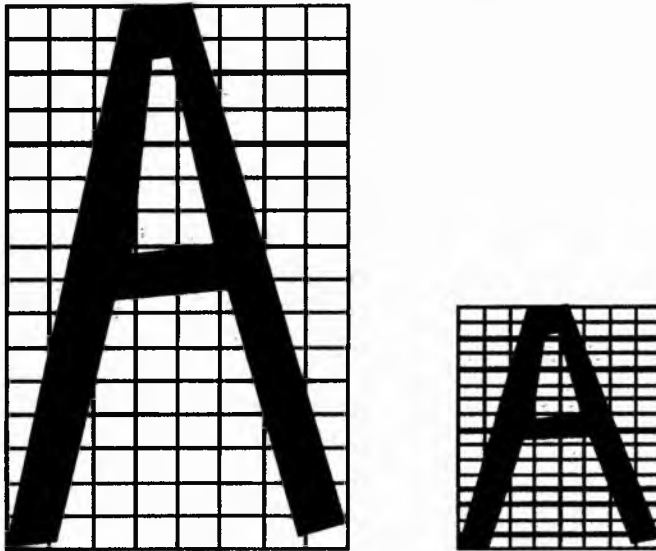


Fig. 13b Size normalisation. Each character area is divided into  $8 \times 16$  small rectangles. Then the values of the pixels in each area are used to calculate an average luminance value for the given rectangle. The number of physical pixels which belong to a small rectangle depends on the physical dimensions of the character and hence it is variable between character sizes.

Due to various character sets used in the number plates, different illumination conditions and different positions of the camera with respect to the car, the differences between various instances of the same character are rather large in spite of various normalisations performed. As a consequence, various instances of the

same character will be spread over a large volume in the input space. The training set contains 180 patterns.

The weights are initialised with random values between -0.5 and 0.5. The momentum parameter is 0.8. Various initial learning rates between 0.5 and 0.01 have been tried for the standard training but the training failed (with a time-out limit of 15,000 epochs). An initial learning rate of 0.5 has been used in the training of the subgoals. This initial learning rate has been gradually decreased during the subgoal training down to values of 0.01.

Because of the practical application which required ensuring a minimal worst-case performance, the termination condition was not imposed upon a sum of squared errors but upon the maximum absolute error value. The termination condition was:

$$\max\_error\_per\_epoch = \max_j \left\{ \sum_{k=1}^{36} |target_{jk} - output_{jk}| \right\} < error\_limit$$

where j indexes the patterns and k indexes the output units.

#### **4.6.1.1. Constraint based decomposition versus standard training.**

The standard approach of training the whole training set is compared with a constraint based decomposition approach. The weight changing mechanism for both approaches is the generalised delta rule [McClelland, 1986].

Many attempts have been done to train the car plate recognition problem with the classical approach of using the entire pattern set and various other values for the training parameters. However, all these attempts failed. In order to allow a meaningful comparison (by using exactly the same initial weight state and parameters), other attempts were performed with the standard approach running in exactly the same conditions as the CBD approach. Only one such paired experiment will be described here.

In each trial, two networks were initialised with the same initial weight state and used the same values for the momentum and learning rate during the whole training process. One network used the classical technique of training with the whole set of patterns and the other was trained with a constraint based decomposition of the training set. Subsequently, the standard training was tried with different parameters (especially learning rate, as described above) but it was never successful.

The standard approach training fails to converge in 15000 epochs (see fig. 14). For larger learning rates, the learning curve presents the same initial drop after which the curve becomes almost flat. For large learning rates (around 0.5 and larger) the saturation phenomenon appears with the weight growing excessively. The CBD training converges to an error limit of 0.3 in approx. 13200 epochs and to an error limit of 0.2 in approx. 13600 (see fig. 15).

As the final performances of the net depend ultimately on the error limit for the last subgoal only, the speed of the training can be dramatically increased if a more relaxed error limit is used to detect the end of a subgoal training. An error limit of approx. 0.75 for the subgoals reduces the total training time (in epochs) by approximately a half (the shadowed areas in fig. 16). This intermediate error limit depends very much on the problem. The graph in fig. 16 was obtained with the same data as that in fig. 15. and is used just to illustrate the possible improvement brought by using a more relaxed error limit for the subgoal training.

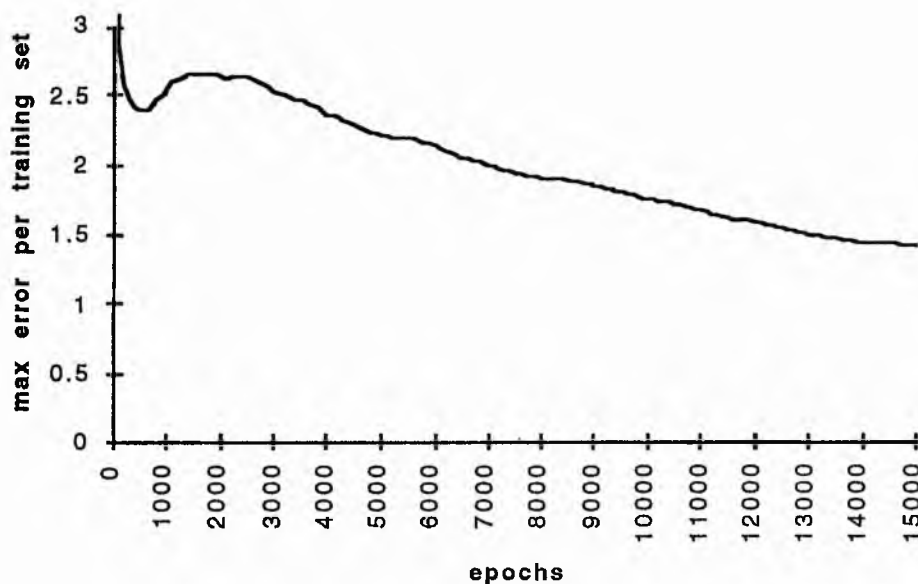


Fig. 14. The evolution of the error (maximum error over the pattern set) during a standard training session. After an initial drop, the error decreases very slowly and the training fails to converge in 15,000 epochs.



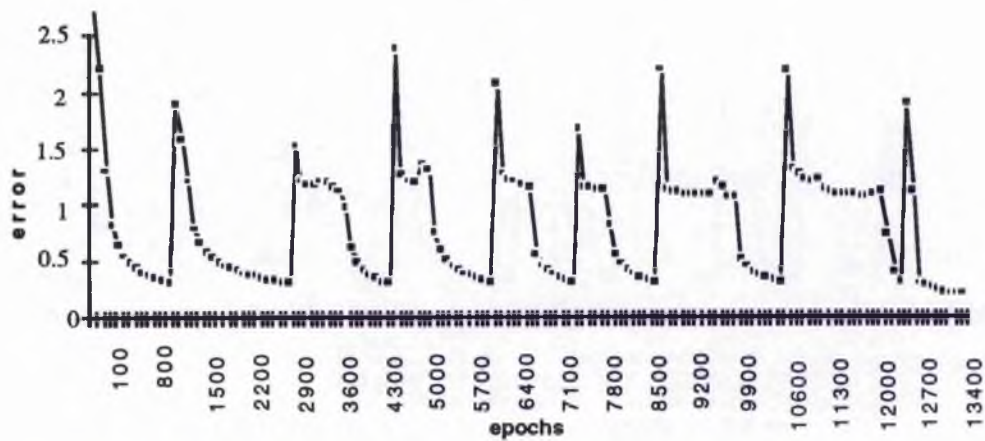


Fig. 15. CBD training. Each peak corresponds to the start of a subgoal training. The sudden increase in error is due to the new patterns.

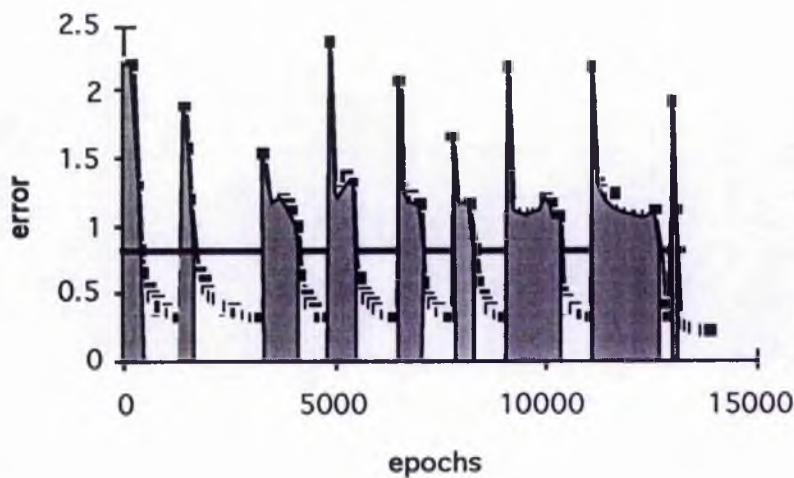


Fig. 16. The reduction of the training time using a larger error limit for the subgoal training. The effective training would be given by the shaded areas.

Note that an epoch for the whole training set necessitates the calculation of the weight changes determined by the entire number of training patterns whereas an epoch for a subgoal training set necessitates only the calculation for the number of the patterns in the subgoal training set. Therefore, the CPU time needed for an epoch in the standard technique will be much longer than the time needed for an epoch for any subgoal but the last one which is the whole training set. As discussed in chapter 2, a better comparison can be performed if the training time is measured

in connection-crossings or operations. In this particular case, the number of operations performed for one weight change is constant and therefore the connection-crossings are equivalent to the number of operations. The batch pattern presentation needs approximately 2.7 million connection-crossings for 180 patterns and 15,000 epochs whereas the CBD pattern presentation algorithm needs approximately 1.2 million connection-crossing (the sum of all subgoal training sessions).

As discussed in chapter 2, the results of the generalised delta rule as a weight updating algorithm can be improved using various techniques. These techniques can also be used with a constraint based decomposition pattern presentation algorithm. It is believed that the use of most of these techniques would not affect essentially the overall result of the comparison.

A strict constraint based decomposition would ask for subgoals formed by adding the characters one by one i.e. the first subgoal is implemented by a training set formed with instances of the first character, the second with instances of the first two characters, etc. This is inefficient because all the units in the output layer whose class is not present in the current subgoal will tend to have zero weights. This is because the initial weights values are small values and the targets of these units are always zero because there are not patterns from the classes which correspond to these units in the pattern set. In this conditions the distance in weight space between the initial position and the solution would be relatively large for the initial subgoals. Without any other precautions, the CBD pattern presentation algorithm would not be able to help the training. For this reason, the first subgoal was built with a pattern from more than one class ensuring that the first subgoal offers a fairer start. The subgoals were defined using 30, 40, 60, 80, 100, 120, 140, 160 and 180 randomly chosen patterns.

#### **4.6.1.2. Investigating the search directed by subgoals**

Fig. 15 shows the evolution of the error during an CBD training session. Note that the error goes up at the beginning of the training of each subgoal but the error does not accumulate from a subgoal to another.

In fig. 17a the evolution of the error during a particular subgoal training is plotted against the number of epochs. Both the mean error over the patterns in the current subgoal and the mean error over the patterns in the previous subgoal are shown. This graph sustains the CBD approach and the importance of the pattern

presentation algorithm. The graphs shows that the information gained during the training of previous subgoal (represented by the mean error over the patterns in the previous subgoal) can be preserved. At the beginning of the subgoal training, when new patterns are added in the training set, the mean current error jumps to higher values (around 0.8 in this graph). However, this jump is due mostly to the new patterns and this is shown by the fact that the mean error over the patterns in the previous subgoal does not increase dramatically.

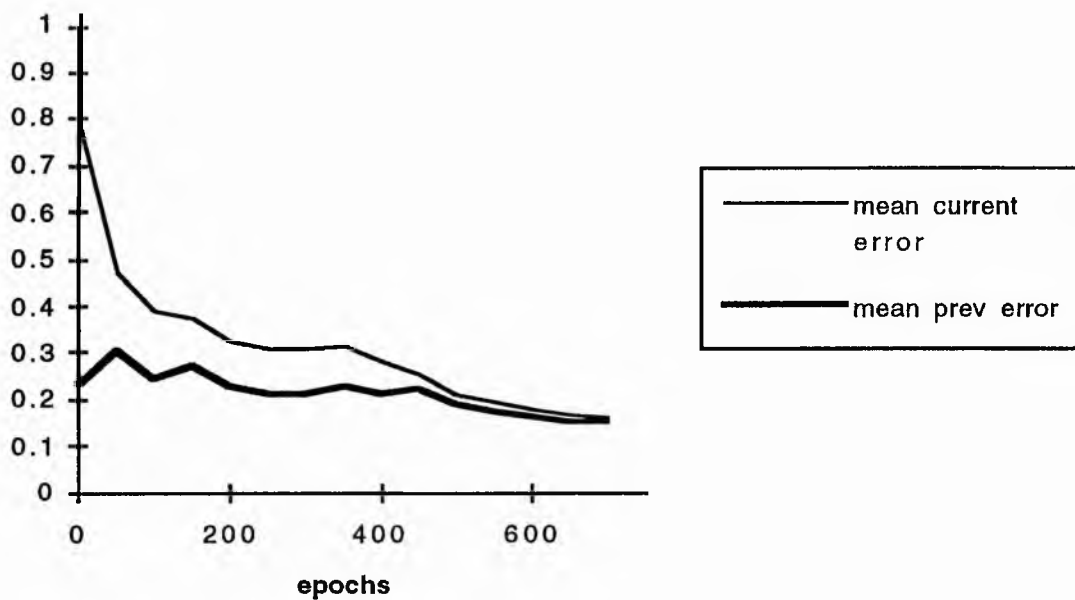


Fig. 17a. The evolution of the error during a subgoal training. The mean errors over the current and the previous subgoals are presented.

In fig. 17b, another subgoal training is presented. This time, the maximum absolute error per current subgoal (which determines the termination) and the number of patterns for which the error is above the error limit are plotted along the mean errors over the patterns in the current and previous subgoals. By error it is meant absolute error. The mean values were obtained dividing the sum of the absolute error of all patterns by the number of patterns. The termination condition is for the maximum error to be less than 0.3.

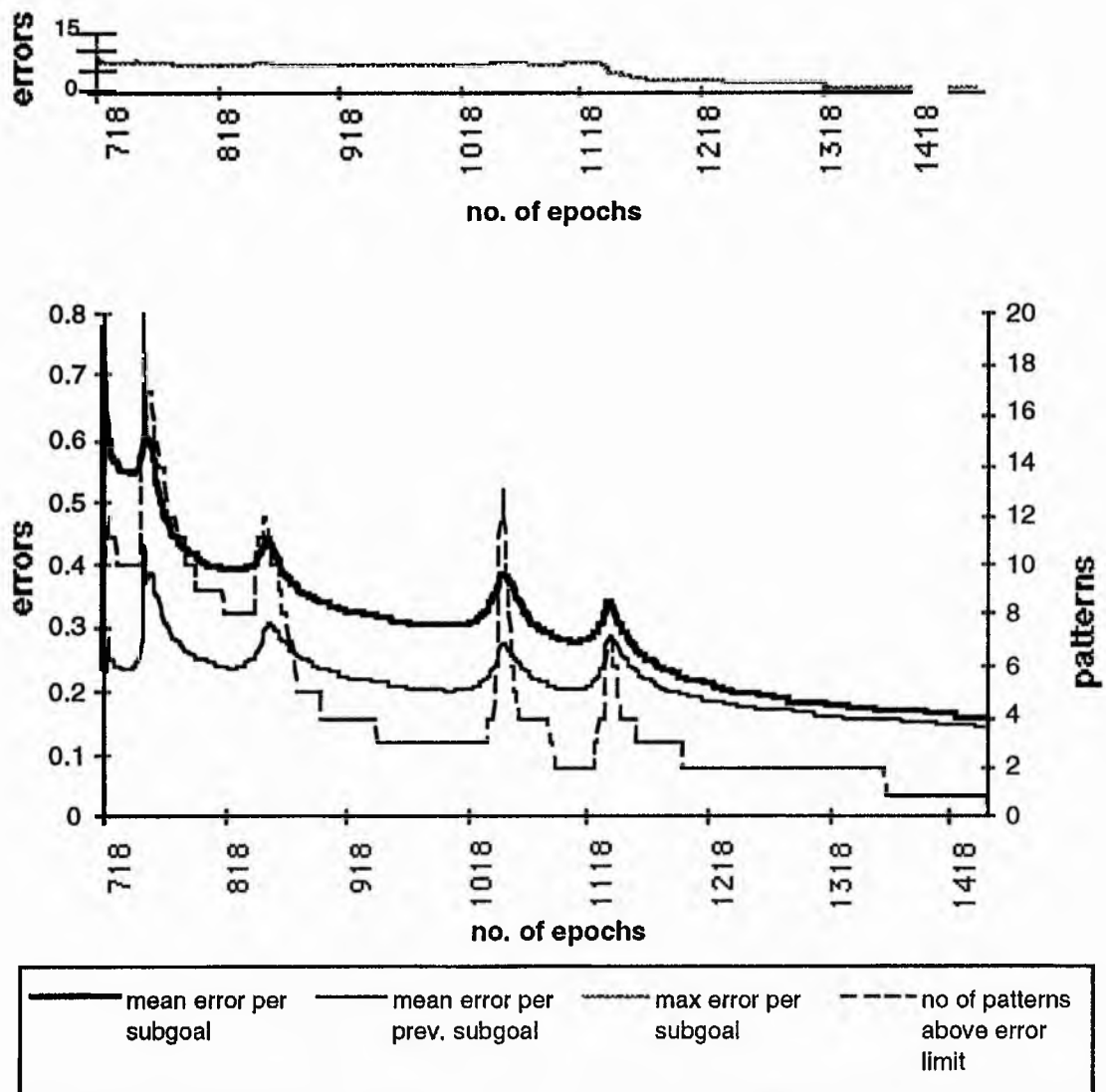


Fig. 17b The evolution of the error during a subgoal training. The mean errors over the current and the previous subgoals, the maximum error over the current subgoal and the no. of patterns above the error limit are plotted.

Let us analyse this graph. Approximately in the 700-th epoch, the previous subgoal training is finished due to the maximum error becoming smaller than the error limit of 0.3. A new subgoal training set is constructed by adding new patterns and a new subgoal training is started. The newly added patterns determine a sudden jump in the mean error per subgoal (from below 0.3 to approximately 0.9) and of the summed maximum error per current goal (from approximately 3 to approximately 15 on the small scale to the left of the graph). At the same time, the number of

patterns above the error limit jumps for the same reason from 0 (the previous training was finished) to 10 because 10 new patterns have been added and all of them are unknown to the network and hence produce high error. The presence of these new patterns determine a slight deterioration of the performance over the patterns included in the previous subgoal: the error over the previous subgoal increases a little but remains much smaller than the error over the new patterns. As the subgoal training proceeds, the summed error for the new patterns decreases slowly from 15. to about 4, value which corresponds to a maximum pattern error below the error limit.

This graph shows that even when the error over the current subgoal training set is large due to the presence of the newly added patterns, the error over the previous subgoal training set remains small which shows that the search takes place in or near the sub-space of the constraint space determined by the previous subgoal. Therefore, in this case, the subgoal manages to direct the search for the solution.

#### **4.6.2 Search restricted by subgoals.**

##### **4.6.2.1 Illustrating the algorithm with some toy-problems.**

The constructive CBD algorithm which implements the search restricted by subgoals has been tested with linearly inseparable problems containing the XOR training set among other desired I/O points. An example is presented in fig. 18. The figure contains both the training set and the hyperplanes the algorithm found in solving the problem.

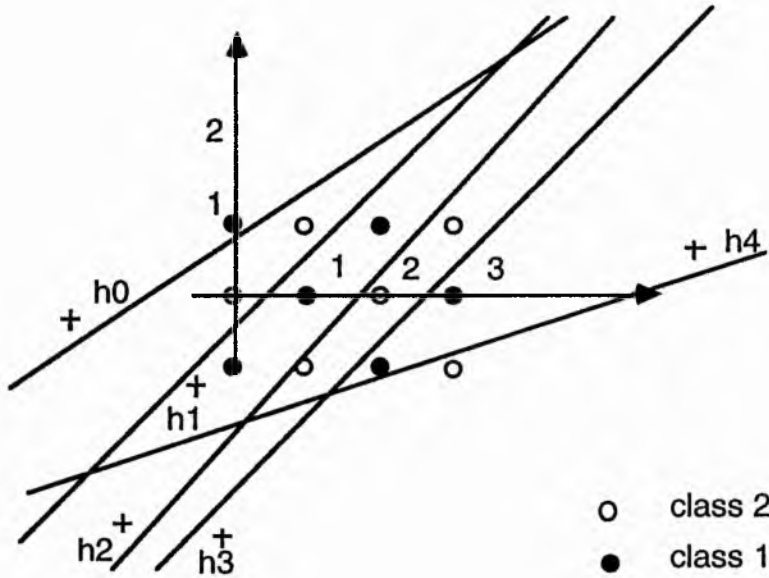


Fig. 18. An example of a I/O set. The architecture obtained at the end of the training uses 5 hyperplanes.

The simple version of the 2-class algorithm (see fig. 4) was used to solve this problem. The architecture obtained at the end of the training used 5 hyperplanes of the form  $w_1x + w_2y + w_{bias} = 0$ .

The solution is interpreted in the following way: a pattern will be classified as C1 if it determines (a positive activation of the neuron associated with  $h_0$ ) **or** (a negative activation of the neuron associated with  $h_0$ ) **and** (a positive activation of the neuron associated with  $h_1$ ) **and** (a positive activation of the neuron associated with  $h_2$ ) **or**...etc. Logical **and** has a higher priority than logical **or**. As previously described, the expressions for C1 and C2 can each be seen as a reunion of regions obtained by intersecting half-spaces determined by different hyperplanes.

The solution is:

$$C1 = h_0 + \bar{h}_0 h_1 h_2 + \bar{h}_0 h_1 \bar{h}_2 h_3 h_4 \quad (3)$$

$$C2 = \bar{h}_0 h_1 \bar{h}_2 h_3 \bar{h}_4 + \bar{h}_0 h_1 \bar{h}_2 \bar{h}_3 + \bar{h}_0 \bar{h}_1 \quad (4)$$

The horizontal bar means the sign of the correspondent hyperplane is minus and the hyperplanes with sign = nil are missing from the expression of the solution.

The net has the three layer structure described in fig. 5. There are 5 neurons on the first hidden layer, each of them corresponding to a hyperplane. There are 6 neurons

on the next hidden layer, three of them corresponding to the three terms in (3) and the other three corresponding to the other three terms in (4). There are two neurons on the output layer, each of them corresponding to a class.

#### 4.6.2.2. Constraint based decomposition versus standard backpropagation

The CBD algorithm has been tested with the 2-spiral problem proposed by Wieland (see [Lang, 1988]). In the following, the standard training set of this problem contains 194 patterns of two classes (see fig. 19). The patterns are distributed along two intertwined spirals which go round the origin 3 times. An alternate dense training set containing 776 patterns but covering the same length of the spirals (3 times round the origin) is also used.

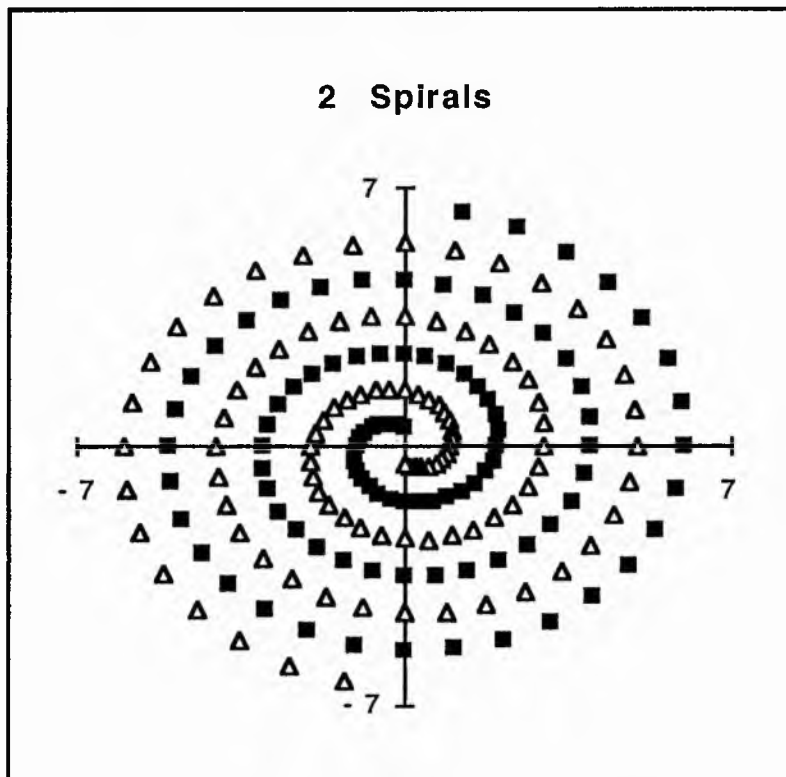


Fig. 19. The standard training set of the 2-spiral problem containing 194 patterns. The triangles are input points for which the output should be 0 (white) and the squares are input points for which the output should be 1 (black).

In [Lang, 1988], three different weight changing mechanisms are used: standard backpropagation [Rumelhart, 1986], cross-entropy backpropagation [Lang, 1988] and quickprop [Fahlman, 1988].

There is an architectural aspect of this comparison which should be discussed. Lang and Witbrock say that *"Initially, we attempted to learn the task with "standard" backpropagation nets, containing few layers of hidden units, and connections only between adjacent layers. These experiments failed, convincing us both of the difficulty of the task, and of the need to design a specific network architecture to suit the problem"*. For these reasons they use an architecture with 19 units on 5 layers and short-cut connections between layers.

The final architecture found by the CBD uses only 3 layers of active weights of which two are very sparse (contain very few connections). This shows that a solution with only few layers and without short-cut connections does exist and the failure of the standard backpropagation reported by Lang and Witbrock is due mainly to the training algorithm and not to the architecture as suggested.

As showed in the chapter on training, a convenient measure of the training speed is the number of connection crossings or the number of operations. In the case of backpropagation training, they are equivalent since the number of operations per epoch and the architecture remain constant during the training. Lang and Witbrock report an average number (over 3 trials) of 20,000 epochs for standard backpropagation in an architecture with 5 layers and 138 connections. This is equivalent to:

$$138 \text{ (connections)} * 2 \text{ (crossings per epoch)} * 194 \text{ (patterns)} * 20,000 \text{ (epochs)} = 1,070,880,000 \text{ operations (connection crossings)}$$

A more compact architecture using only 4 layers and 68 connections is reported to need 60,000 epochs to learn the same task. In operations, this is equivalent to:

$$68 \text{ (connections)} * 2 \text{ (crossings per epoch)} * 194 \text{ (patterns)} * 60,000 \text{ (epochs)} = 1,583,040,000 \text{ operations (connection crossings)}$$



The CBD algorithm in a version using perceptron training, with redundancy check and locking detection and without linear separability checks<sup>2</sup> reported an average number of 9,654,852.2 operations (connection crossings) over a set of 20 trials with the standard 2-spiral pattern set containing 194 patterns. This is equivalent to an speed increase of about 100 times over standard backpropagation in the dedicated architecture used by Lang and Witbrock. The average number of hyperplanes (units in the first hidden layer) used in the solution built by the CBD algorithm was 53.65.

With an LMS weight changing mechanism instead of perceptron training, the CBD algorithm reported an average number of 11,979,130.4 operations (connection crossings) over a set of 20 trials with the standard training pattern (194 patterns). The average number of hyperplanes used in the solution was 53.7.

#### **4.6.2.3. Constraint based decomposition versus divide and conquer network (DCN)**

These experiments compared the performance of the CBD algorithm with the performance of DCN networks (see Chapter 2, section 2.3.3 and [Romaniuk, 1993]) on the 2-spiral problem.

Unfortunately, some of the problems regarding the speed comparison between two different training algorithm discussed in the chapter on training are present here. DCN, as CBD constructs the network during the training. In this conditions, the number of epochs is not very useful unless accompanied by the description of the architecture during each epoch which would be very tedious. Indeed, Romaniuk and Hall in their paper, do not use the number of epochs or connection crossings to give information about the speed but the running time on a specific machine, a Sparcstation 2. Unfortunately, this particular machine was not available for tests with CBD so the comparison is not straightforward.

---

<sup>2</sup> The redundancy check does not affect the number of connection crossings. The locking detection improves the speed performances so it is expected that a simple version of CBD would perform slightly worse from the point of view of the speed. However, even the simplest version of CBD would perform with a speed of the same order of magnitude. For more information about these alterations of the basic algorithm see chapter 5 Enhancements of the CBD approach.

In order to compare the results, a correction coefficient has been calculated to take into account the speed differences of the hardware used. This coefficient has been calculated in the assumptions that: i) the running time is mostly spent performing floating point operations ii) the differences due to different implementations/compiler are negligible. In these conditions and taking into considerations the floating point specifications of the Sparcstation 2 (SpecFP92 - 17.0) and Sparcstation 10/41 (SpecFP92 - 68.8) the correction factor is 3.98 i.e. if an algorithm runs in  $t$  seconds on the Sparcstation 2 it will run in approximately  $t/3.98$  seconds on the Sparcstation 10/41. It must be stated that both the assumptions and the use of the SpecFP92 figures to calculate such a correction factor are arguable and the results of the comparison must be considered only as indicative results.

All trials with the CBD algorithm were performed on a Sun Sparcstation 10 and the trial time refers to CPU time. A pattern set containing 194 patterns of the 2-spiral problem was learned in an average of 56.35 seconds (average over 20 trials). This compares with an average of 772.1 seconds (averaged over 10 trials with improvement limits from 100 to 1000) for the DCN algorithm run on a Sparcstation 2. Using the correction coefficient calculated above, this reduces to 193.99 corrected seconds on Sparcstation 10/41. From these figures, it follows that the implementation of the CBD algorithm is approximately 3.44 times faster than DCN on this particular problem.

CBD solved the problem with an average of 53.65 hyperplanes (hidden units on the first layers) over 20 trials. The CBD architecture needs approximately another 20 units in the second hidden layer and 1 unit in the output layer. DCN solves the problem with an average of 34.6 units (averaged over the same trials as above) distributed on an average of 8.6 layers (minimum 6 layers, maximum 12 layers). It is apparent that the more complicated network architecture (multilayer with short-cut connections) used by DCN allows the use of fewer units.

## **4.7. Conclusions and discussion**

### **4.7.1 Conclusions of the experiments**

The experiments investigated the ideas of the constraint based decomposition approach.

The experiments presented in section 4.6.1 show the importance of the pattern presentation algorithm. Both the standard training and the CBD training used the same weight change mechanism: the generalised delta rule. However, the standard training fails systematically on this problem whereas the CBD can be successful. However, the success of the pattern presentation algorithm itself depends too much on the subgoal definition. Care is needed in the definition of the first subgoal (so that samples from all regions of the input space are present in the initial subgoal) and in the definition of subsequent subgoals. A balance should be maintained between the amount of information brought by a new subgoal and the learning parameters (learning rate, momentum, etc.).

Although in this case there was an improvement, the CBD pattern presentation algorithm by itself does not guarantee success because an unsuitable definition of the subgoals can determine the failure of a subgoal which would mean that all the previous subgoal training sessions have been wasted. The failure of a subgoal training towards the end of the subgoal chain would put the network in a situation similar to that of the standard training: to learn all or almost all patterns starting from a weight state which is not able to respond correctly to many patterns.

The CBD pattern presentation algorithm can be used with any weight changing mechanism. Combined with backpropagation it can give an improvement over the standard training but the results are not always guaranteed.

The experiments presented in section 4.6.2 illustrated the search restricted by subgoals version of the constraint based decomposition approach. Firstly, the CBD algorithm is illustrated on a toy-problem whose pattern set includes XOR's patterns. Subsequently, the performance of the algorithm is compared with that of the standard backpropagation approach and with that of another very recent constructive algorithm, divide and conquer networks (DCN).

CBD compares very favourably with the standard backpropagation. A fully connected multilayer architecture with a standard batch pattern presentation algorithm fails to train the 2-spiral problem (see [Lang, 1988]). Lang and Witbrock's dedicated multilayer architecture with short-cut connections between layers manages to train the pattern set but CBD is shown to be much faster and more reliable.

CBD gives comparable performance with that of the DCN in terms of reliability since both algorithms are guaranteed to find a solution. CBD compared favourably

with DCN in terms of speed. Although an accurate speed comparison could not be performed, reasonable approximation suggests that CBD is a few times faster on problems with the degree of complexity of the 2-spiral one. The architecture constructed by CBD uses only 3 layers of active weights whereas the architecture constructed by DCN varies and uses a number of layers between 7 and 12. However, DCN is able to use the flexibility given by the more complicated architecture and in general, solves the problem with fewer units.

#### **4.7.2 The characteristics of the CBD architecture and training algorithm**

The characteristics of the CBD architecture and training algorithm (which implements the search restricted by subgoals) are:

1. The CBD training algorithm is composed of a weight updating algorithm for a single layer (delta rule, for instance), the CBD pattern presentation algorithm and the CBD construction method.
2. The CBD network has the abilities of a multilayer perceptron but the training is performed exclusively in subnets with a minimal architecture containing only one layer and one neuron. This is the simplest possible training problem from the point of view of the architecture (the best possible situation for the first two factors discussed in the chapter on training: only one layer so one can use a simple weight update rule and only one neuron so that number of dimension of the weight space is minimum).
3. CBD trains exclusively training sets with  $n$  examples of which  $n-1$  are already correctly classified. This is the simplest possible training problem from the point of view of the training set and this eliminates the herd effect (useless competition between different hidden units to implement the same feature of the input patterns). Furthermore, all the training sets (barring one) have fewer patterns than the original set and most of them have only very few patterns. This is a reduction of the training set's dimensionality.
4. CBD finds automatically an architecture able to solve the training problem. The algorithm guarantees the absence of useless units (whose outputs are not actually used in performing the classification). The architecture found by the net is sometimes the minimal one but the algorithm does not offer guarantees in this sense. However, the convergence is guaranteed.

5. The computation involved in training is very simple. No first or second order derivatives are used. No pre-processing is needed. The training is very fast and the resulting network is able to solve linearly inseparable tasks.

6. The fact that the first hidden layer is not fully connected to the **and** layer avoids the interference between hyperplanes which is one of the difficulties faced by a fully connected net.

7. CBD can be used for incremental learning, in which a trained network is asked to adapt itself to new patterns. The CBD net will train only the smallest possible region(s) of the input space which contain the new pattern(s). The hyperplanes introduced to satisfy the new patterns will not affect the classification of other regions.

#### 4.7.3 Relation to other work.

In this section, some differences between CBD and other related techniques will be discussed.

First of all, it must be emphasised that the constraint based approach must not be identified with the particular constraint based algorithm presented. The constraint based decomposition approach is a variation of the divide and conquer principle in which the training is seen as a constraint satisfaction problem and the division of the problem into sub-problems is based on these constraints.

The constructive CBD algorithm is just a particular implementation of the CBD approach. This particular implementation has various elements in common with other techniques and these elements will be discussed in the following.

The CBD algorithm can be seen as building a decision tree. The entropy nets of Sethi (see [Sethi, 1990a], [Sethi, 1990b]) use a decision tree to classify the regions and two layers of weights, one for logical **and** and one for logical **or**. These layers are similar to those used by CBD. However, the building of the decision tree can be a very lengthy process because it involves testing very many candidate questions for each node in the tree. For instance, CART (Classification and Regression Trees) uses a standard set of candidate questions with one candidate test value between each pair of data points. A candidate question is of the form  $\{Is\ x_m \leq c\}$  where  $x_m$  is a variable and  $c$  is the test value for that particular variable. At each node, CART searches through all the variables  $x_m$ , finding the best split  $c$  for each. Then the best of the best is found (see [Breiman, 1984]). For a problem in a high

dimensionality space and many input patterns, this can be a very time consuming process. On the other hand, the techniques which build a network by converting a decision tree offer some intrinsic optimisation. Usually, in the process of building the tree, some measures are taken to ensure that the splits optimise some factors such as the information gain.

CBD builds up the desired I/O surface gradually, one region after another. The idea of locally constructing the I/O shape is present in all radial basis function (RBF) algorithms (see [Broomhead, 1988], [Moody, 1989], [Musavi, 1992], [Poggio, 1990]). In RBF's case, one unit with a localised activation function will ensure the desired response for a small region of the I/O space. However, there are situations in which a net building piece wise linear boundaries is better than an RBF net. Furthermore, for an RBF net to be efficient, a pre-processing stage must be performed and parameters such as radii of the activation functions, their shape and orientation, the clustering, etc. must be calculated. By contrast, CBD is relatively simple.

An RBF network will respond only for inputs which are close to the inputs contained in the training set. For completely unfamiliar inputs, the RBF network will remain silent automatically signalling its incompetence. At the same time, the CBD network (as any other network using hyperplanes) automatically extends their trained behaviour to infinity and gives some response for any input no matter how unfamiliar. The potential problems introduced by such behaviour can be eliminated by using the techniques for validating individual outputs discussed in Chapter 2. If the chosen validation method signals that the input is far from the inputs which were used during the training, the output will be ignored. Thus, the behaviour of the network using hyperplanes will be similar from this point of view to the behaviour of an RBF network.

CBD builds up the solution by combining the solutions of different subgoals. The idea of building the solution by combining partial solutions was proposed by Hinton and Anderson in [Hinton, 1981]. However, the combining method proposed there is a simple sum of the weight matrices and it works only for orthogonal patterns. This can be seen as a particular case of CBD in which a constraint is one pattern. In this special case, each subspace of the constraint space is characterised by a unique weight state (a partial solution). The set of partial solutions can be combined to give a unique weight state which satisfies all the constraints and therefore is the solution. In constraint space, the above technique is

equivalent to finding the subspaces corresponding to each pattern and directly calculating their intersection which is the solution.

The ideas of training only one neuron at a time and gradually building the net are present in the Cascade Correlation (CC) net proposed by Falhman and Lebiere in [Falhman, 1990]. However, CC algorithm uses the whole pattern set and the resulting architecture is different. CC builds feature detectors which could be useful in some problems. The advantage of using the whole training set is that the solution can be optimised from some point of view. In CC's case, the weights are chosen so that the correlation between the output of the last added unit and the output is maximum. This ensures that the unit is as useful as possible. CC, as for any other global optimisation algorithm cannot work on-line. For the optimisation to be effective, the algorithm must restart from scratch each time a new pattern is added. Otherwise, the solution given by such an algorithm could be as bad as the solution given by local optimisation algorithms such as vanilla CBD. The lack of global optimality is the price paid by vanilla CBD for its on-line capabilities. On the other hand, if one takes into consideration the fact that the derivation of the optimal decision tree is NP-complete (see [Hyafil, 1976] cited by Golea and Marchand in [Golea, 1990]), this price does not seem too high.

There are few algorithms which ensure the convergence of the training process. The upstart algorithm [Freen, 1990] builds a hierarchical structure (which can be eventually reduced to a 2 layer net) by starting with a unit and adding daughter units which cater for the misclassifications of the parents. Sirat and Nadal proposed a similar algorithm in [Sirat, 1990] However, both of them work for on/off units only. Mezard and Nadal in [Mezard, 1989], proposed a tiling algorithm which starts by training a single unit on the whole training set. The training is stopped when this unit produces the correct target on as many patterns as possible. This pseudo-solution weight state is given by the Gallant's pocket algorithm ([Gallant, 1986]) which assumes that if the problem is not linearly separable the algorithm will spend most of its time in a region giving the fewest errors.

The pocket algorithm simply monitors the weight change and stops the training after some chosen time  $t$ . The choice of the time-out limit  $t$  can be important. A large  $t$  leads to a slow training, a small  $t$  could prevent reaching a reasonable solution.

The tiling algorithm has a very inefficient pattern presentation algorithm. It is very inefficient to start with the whole training set because most of the time (for any interesting problem) the training will fail. In contrast, CBD starts always with a

simple problem for which the existence of the solution is guaranteed and increases gradually the complexity of the problem. The pocket algorithm does not offer any guarantees regarding the optimality of the weight state obtained in any finite time.

Romaniuk and Hall [Romaniuk, 1993] proposed a divide and conquer net which builds up the network. Their divide and conquer strategy starts with one neuron and the entire training set. If the problem is linearly inseparable (which is the usual situation), the first training is bound to fail and this is detected by a time-out condition. In comparison, CBD starts with the minimum problem which is guaranteed to have a solution. The divide and conquer technique has a more complicated pattern presentation algorithm. This pattern presentation algorithm also requires a pre-processing stage in which the nearest neighbour is found for each pattern in the training set.

The architecture given by the divide and conquer algorithm is similar to that of a Cascade Correlation network, with each unit connected to all the input units. However, the architecture of the DCN network depends on the initial weight state which can be inconvenient in some cases.

The extenatron proposed by Baffes and Zelle in [Baffes, 1992] grows multilayer networks capable of distinguishing non-linearly separable data using the perceptron rule for linear threshold units.

The extenatron looks for the best hyperplane relative to the examples in the training set. If the problem is not completely solved by this best hyperplane, a new unit is added. This unit is connected to the inputs and to all the previous units. Thus, the dimension of the problem's space is extended and the problem could become linearly separable. In the worst case, each unit will separate a single pattern but experiments showed that this doesn't happen unless the problem is "pathologically difficult" (the two-spiral problem is quoted as such a problem). Experiments showed CBD is far more efficient than this even in solving the two-spiral problem.

The extenatron algorithm assumes theoretically that the perceptron training finds the best possible hyperplane. In practice, this is not always the case. A maximum number of epochs is used to detect the linear inseparability of the problem. In the first instance, the unit is trained so that the classification is better than the previous layers' one. Subsequently, the unit is trained until a number of epochs equal to `max_epochs` passes without any improvement. Strictly speaking, this does not ensure that the hyperplane found is the best.



The cascade connection of the extenttron architecture means that, for highly non-linear problems such as 2-spirals, the last few hidden units will have to solve a problem in a high dimensional space (2 dimensions of the input space plus  $n$  dimensions of the first  $n$  hidden units). Although it is a perceptron training regime, the training will be more difficult because of the possible large number of dimensions.

The extenttron was originally presented for binary outputs. An extension to continuous values could couple the extenttron with a backpropagation stage in which the weights could be adjusted to obtain the desired values. In this case, the large number of layers generated by the extenttron is an important disadvantage for backpropagation. The architecture generated by the CBD algorithm which contains always the same number of layers does not have this problem if the network is to be subsequently trained with backpropagation.

Furthermore, if the extenttron architecture was to be implemented in hardware, synchronisation problems might arise due to the existence of paths with very different lengths between input and output. This problem is not present in the case of the solution generated by CBD for which all paths from input units to output units have equal length.

# CHAPTER 5

## Enhancements of the Constraint Based Decomposition

### 5.1 Introduction

In this chapter, some possibilities for improving the training speed of the 2-class training of the constructive CBD algorithm and a connection with a symbolic artificial intelligence technique are discussed. Some other characteristics such as the weak dependence on the initial weight state and the possibility of influencing the generalisation are also discussed.

In section §5.2, possible approaches to improving the training speed are discussed in brief. Sections §5.3 and 5.4 present in details two such approaches: improving the training speed through locking detection and eliminating problems beyond the possibilities of the current architecture. Section §5.5 discusses issues related to the efficient use of the hyperplanes and the elimination of redundant hyperplanes. Section §5.6, analyses the generalisation properties of the constructive CBD algorithm and means to improve them. Other issues as optimality and dependence on the initial weight state are discussed in section §5.7.

### 5.2 Improving the training speed

One of the characteristics of the constraint based decomposition algorithm is that the actual weight updating is performed only in very simple networks with just one non-input neuron. In geometrical terms only one hyperplane is moved at any single time. There are two qualitatively distinct training situations.

The first situation is that of a first training after a new neuron has been added. In this situation, the pattern set contains patterns which form a linearly separable problem and the problem can always be solved. This is because the number of the patterns is restricted to at most  $n$  in  $n$  dimensions and the patterns are assumed to be in general position (i.e. not belonging to a sub-space with fewer dimensions).

The second situation is that of adding a pattern to a training set containing more than  $n$  patterns where  $n$  is the dimensionality of the input space. In this case, the problem is to move the existing hyperplane so that even the last added pattern is correctly classified. There is no guarantee that a solution exists for this problem because the last pattern could have made the problem linearly inseparable. This determines a

termination problem. When should the training be stopped if the error will never go below its error limit?

The simplest solution is to use a time-out condition. The training is halted if no solution has been found in a given number  $N$  of iterations. This condition is used by the simplest implementation of the CBD algorithm (the standard CBD) and by the vast majority of constructive algorithms like the extention [Baffes, 1992], the pocket algorithm [Gallant, 1986] and even the very recent divide and conquer networks [Romaniuk, 1993].

If this condition is used, the choice of  $N$  is crucial for the performance of the algorithm. A large  $N$  will mean that the algorithm could spend a long time trying to solve problems which do not have a solution and this will dramatically affect the total training time. A small  $N$  will cause many training sub-sessions (subgoal sessions for CBD) to be declared as insoluble even if they have a solution. For CBD, this second situation will result in the use of a large number of hidden units and a fragmentation of the global solution which can be undesirable for some problems. An excessively small  $N$  will have negative effects upon the overall I/O mapping, the training time or both, for all algorithms which use this termination condition. Unfortunately, the number of iterations required is not the same for all training sessions and cannot be decided a priori. Some heuristics are needed to ensure that i) most of the training problems which have a solution will be solved and ii) not too much time (ideally no time at all) will be spent with those problems which cannot be solved. These heuristics will be discussed for CBD here but can be easily extended to other training algorithms which use the same termination condition.

### **5.3. Improving the training speed through locking detection.**

An idea for such an heuristic is given by the behaviour of the hyperplane during the training. Let us suppose that after a new unit has been added the first subgoal for the training of the new unit contains only two patterns. This problem can always be solved and the hyperplane can be situated anywhere as long as the two patterns of the initial subgoal are separated. Subsequently, as new patterns are added to the training set, the position of the hyperplane will be changed so that if possible the new patterns are correctly classified as well. There are situations in which the existing patterns determine the position of the hyperplane (within a given tolerance). In these situations, the hyperplane cannot be moved any more (outside the tolerance) without misclassifying some existing patterns. Weir and Polhill in

[Weir, 1993] use the term "locking" to describe a similar situation. The term is very intuitive and will be used throughout this thesis.

In a locking situation, the training can be stopped because considering more patterns will not be able to improve the position of the hyperplane. A straightforward check of the remaining patterns can be done. These remaining patterns can be correctly classified or misclassified by the current position of the hyperplane. The latter ones will be taken care of by new hidden units.

The important improvement brought by this approach is that the training to the time-out condition of those situations in which the hyperplane cannot be moved any further is avoided.

### 5.3.1 Characterisation of the locking situations

#### 5.3.1.1 Two dimensions.

In a two dimensional space, a locking situation can be determined by only 3 points from two classes. An example is presented in fig. 1

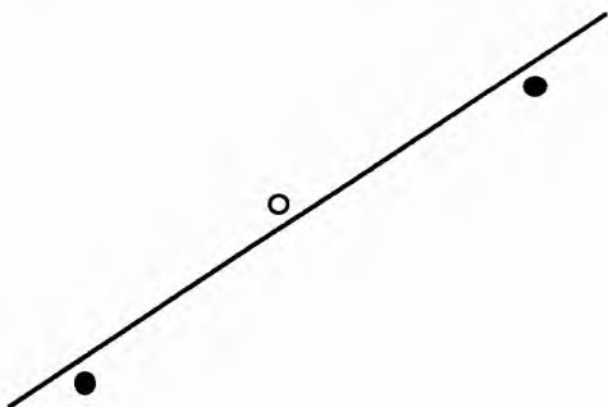


Fig. 1 A locking situation in a two dimensional space. The position of the hyperplane cannot be changed outside the tolerance determined by the points.

A similar example is given in [Weir, 1994a] to illustrate the term of locking. A neural implementation of Mitchell's technique is proposed in [Weir, 1994b] and [Weir, 1993]. In these references, the idea of locking is used in a framework derived from Mitchell's technique. However, the analysis that follows of the points and associated conditions that must be present at locking, how a tolerance can be set and how the notion can be extended are novel.

For any given three points in a locking position like that in fig. 1, the area in which the hyperplane can move without changing the classification of any of the points is presented in fig. 2. Any black point in area C1 or any white point in area C2 will be inconsistent in the sense that the problem will become linearly inseparable and no position of the hyperplane will classify correctly all patterns. Any one additional point (from either class) in the grey area in the figure can either already be correctly classified or can need an adjustment of the position of the dividing hyperplane but a solution will always exist. This area will be called the unconstrained area because any point in it can be put in either class by a suitable modification of the position of the hyperplane. At the same time, this area contains all possible positions of the boundary which will still classify correctly all patterns. If the position of the boundary is seen as equivalent to a concept in symbolic AI and the training is seen as a concept learning, this area can be seen as a version space, a space containing all possible versions of the concept (see section 5.3.2 in this chapter and [Mitchell, 1977], [Mitchell, 1978]). The unconstrained area is formed by the two acute angles

determined by  $d_1$  and  $d_2$  and the triangle determined by the patterns (see the grey area in fig. 2). The locking situation is simply a situation in which the unconstrained area becomes very small i.e.  $d_1$  and  $d_2$  have almost the same slope. In symbolic AI terms, the version space becomes so small that one could see the convergence as being reached. How small this area should become for the locking to be declared depends on the chosen tolerance which in turn depends on the problem.

In a locking situation, any further training can be avoided because the position of the hyperplane cannot be changed anymore (outside the chosen tolerance) without misclassifying some patterns. A simple inspection of the untrained patterns will divide them into two categories: patterns which are correctly classified by the current position of the hyperplane and patterns which are misclassified. These will be called redundant patterns and inconsistent patterns respectively. The terms redundant and inconsistent refer to the current position of the dividing hyperplane. Some patterns are redundant because they do not contribute in any way in determining the current position of the hyperplane. Some other patterns are inconsistent because they do not agree with the partitioning of the space determined by the current position of the hyperplane.

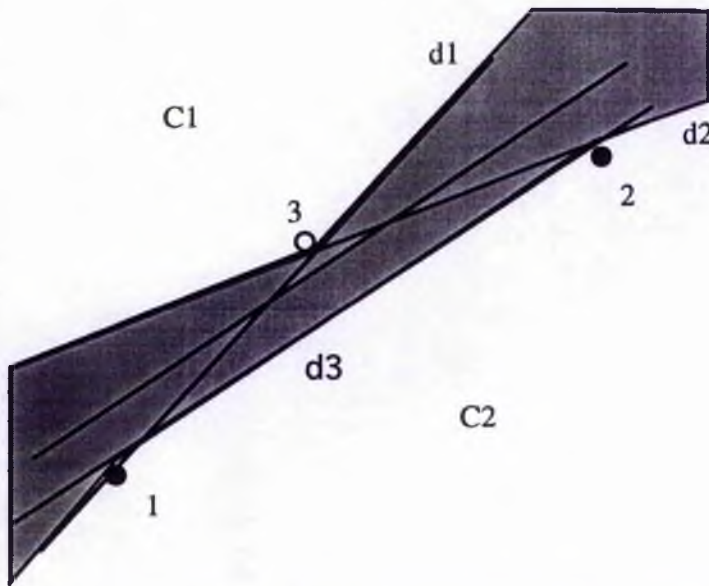


Fig. 2 The tolerance of a locking situation is determined by the slopes of the extreme positions which still classify correctly all the patterns. The slope of any boundary line which correctly classifies all patterns will be in-between the slopes of d1 and d2.

Using this idea, the training can be modified so that after each training, a check is performed to see if locking has occurred. If the hyperplane is not locked, another pattern will be taken into consideration and the training will proceed normally. If the hyperplane is locked and if not all patterns have been separated, a new unit must be added. However, the algorithm enhanced with this locking detection will not spend time trying to solve a linearly inseparable problem with an architecture unable to solve it.

A generalisation of locking for  $N$  dimensions is needed. Let us analyse the 2D case presented in fig 2 and try to identify the elements which characterise such a locking position. Once identified, these elements will help defining the locking position in an  $N$  dimensional space.

The extreme positions of the dividing line so that all patterns are still correctly classified are infinitely close to the edges of the triangle formed by the 3 patterns. However, not any combination of two points from C1 and one point from C2 determine a locking position. Intuitively, the important conditions which characterise the type of locking situation in fig. 2 seem to be i) the fact that the projection of the white pattern on the dividing line falls in-between the projections of the black patterns on the same line and ii) the distance from the patterns to the



dividing line is small. Fig. 3 shows a situation in which the condition ii) is satisfied but the condition i) is not. In this situation, even though the patterns are very close to the dividing line, the unconstrained area (grey in fig. 3) is very large and the locking is not present. Fig. 4 shows a situation in which the condition i) is satisfied but the condition ii) is not. In this situation, even though the projection of the white pattern falls in-between the projections of the two black patterns, the locking is not present. These examples show that the conditions i) and ii) above are both necessary. The next question is whether they are sufficient as well. In other words, if the conditions i) and ii) above are satisfied, is the locking present?

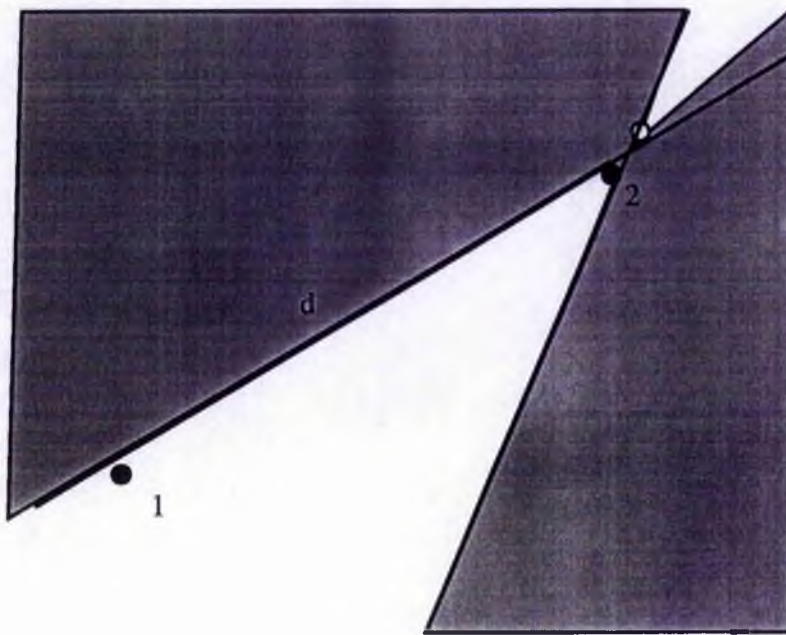


Fig. 3 A non-locking situation. All patterns are very close to the dividing line but the projection of the white pattern falls outside the segment determined by the projections of the black patterns.

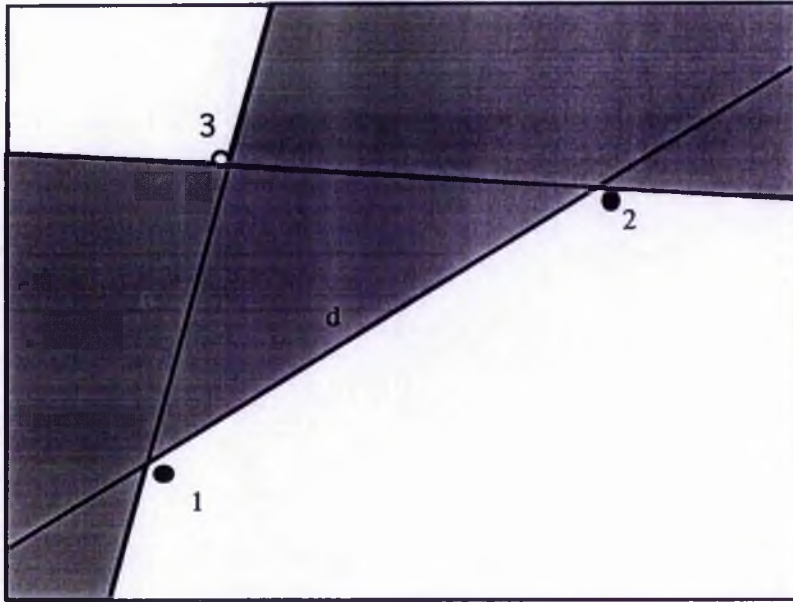


Fig 4. A non-locking situation. The projection of the white pattern falls in-between the projections of the black patterns but the distance from the white patterns to the dividing hyperplane is too large.

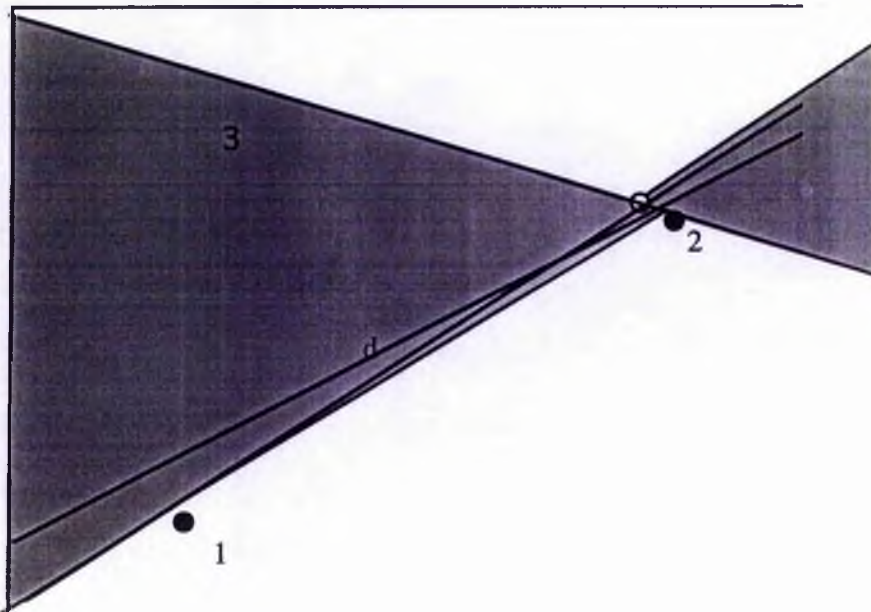


Fig 5. A non-locking situation. Both conditions are satisfied (all patterns are close to the dividing hyperplane and the projection of the white pattern falls in-between the projections of the black ones) but the locking is not present. The conditions i) and ii) are necessary but not sufficient.



Fig. 5 shows a situation in which both conditions i) and ii) are satisfied but the unconstrained (grey) area is still very large and therefore, the locking is not present. The conclusion is that the conditions i) and ii) are necessary but not sufficient. Another element is necessary to characterise the locking situation.

Let us analyse again the locking situation in fig. 1 and 2. Let us imagine moving the white pattern on a direction parallel with the dividing hyperplane (therefore remaining at the same distance). The unconstrained area increases as the pattern comes near the ends of the segment determined by the black patterns. Let us follow the boundaries of the grey area which mark the extreme positions a hyperplane can take so that all patterns are still correctly classified. These boundaries ( $d_1$ ,  $d_2$  and  $d_3$  in fig. 2) will be called the extreme hyperplanes. The slopes of the edges of the triangle determined by the patterns are infinitely close to the slopes of the extreme hyperplanes.

Therefore, the factors which limit the area in which the boundary can be moved are the angles of the triangle determined by the 3 patterns. A locking position appears when the triangle is squashed with a large angle in the corner of the white pattern and very sharp angles in the other two corners. Unfortunately, this observation cannot be generalised easily to more than two dimensions. A more useful way of describing the same phenomenon is to say that the slopes of the lines which form the triangle are very close to each other.

A definition of the locking situation can be given now. Intuitively, a locking situation is a situation in which the position of a potential dividing hyperplane is very much restricted. Using the observation that all the dividing hyperplanes have the slopes between the slopes of  $d_1$  and  $d_2$  in fig. 2, the following definitions come naturally.

**Definition. Extreme hyperplanes.**

Let  $C_1$  and  $C_2$  be two linearly separable sets of patterns from two classes.

Let  $D(x,y)=ax+by+c$  such that

$$i) D(p_i) > 0 \text{ for all } p_i \text{ in } C_1$$

$$ii) D(p_i) < 0 \text{ for all } p_i \text{ in } C_2$$

$$iii) \text{ For any other } d_i \text{ satisfying i) and ii), } \frac{\partial d_i}{\partial x} \leq \frac{\partial D}{\partial x}.$$

Let  $d(x,y)=dx+ey+f$  such that

i)  $d(p_i)>0$  for all  $p_i$  in  $C1$

ii)  $d(p_i)<0$  for all  $p_i$  in  $C2$

iii) For any other  $d_i$  satisfying i) and ii),  $\frac{\partial d_i}{\partial x} \geq \frac{\partial d}{\partial x}$ .

The hyperplanes  $d$  and  $D$  will be called the extreme hyperplanes of  $C1 \cup C2$ .

The conditions i) and ii) mean that a hyperplane satisfying them is a dividing hyperplane. The conditions iii) mean that among all the dividing hyperplanes,  $d$  has the smallest slope and  $D$  the largest one.  $D$  and  $d$  correspond to the extreme hyperplanes  $d_1$  and  $d_2$  in fig. 2.

**Definition. A locking situation with  $\epsilon$  tolerance.**

Let  $C1$  and  $C2$  be two linearly separable sets of patterns from two classes and  $d$  and  $D$  the extreme hyperplanes of  $C1 \cup C2$ .

The patterns in  $C1$  and  $C2$  are said to create a locking situation with tolerance  $\epsilon$  if and only if:

$$\left| \frac{\partial d}{\partial x} - \frac{\partial D}{\partial x} \right| < \epsilon$$

The definition says that a linearly separable set of patterns from two classes determine a locking situation with tolerance  $\epsilon$  if and only if the difference between the largest and the smallest slope of a possible dividing hyperplane is smaller than  $\epsilon$ .

It must be said as an observation that the tolerance  $\epsilon$  is essential in the sense that for any linearly separable set of points from two classes  $C1$  and  $C2$ , there exists a tolerance  $E$  such that the  $C1 \cup C2$  create a locking situation with tolerance  $E$ . In order to prove this it is necessary only to prove that for any two linearly separable sets of points the extreme hyperplanes exist (proof given in the linear separability section, 2D case) and to use the fact the set of real numbers is unbounded (i.e. for any slope difference  $\Delta e$ , there exists an  $E$  so that  $\Delta e < E$  and therefore, the situation is a locking situation with tolerance  $E$ ). Because of this observation, the tolerance  $\epsilon$  will assumed to have a value  $\epsilon_p$  suited to the problem. The locking will be said to

be present if the tolerance of the locking situation is reasonably small (smaller than  $\epsilon_p$ ) and said to be absent otherwise.

Although the definition was intuitively inspired and justified by the locking situation presented in fig. 2, its scope is not limited to this situation. Other locking situations of a different type from the one presented in fig. 2, are also covered. An example of such a different locking type is presented in fig. 6. Note that if the pattern set contained only three (any three) of the four patterns, the locking would not be present.

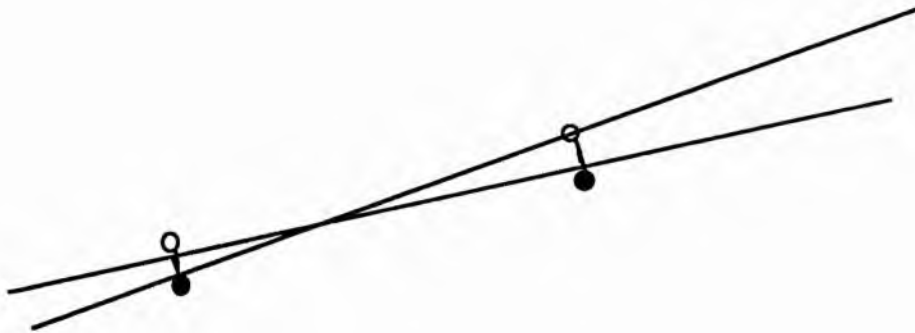


Fig. 6 A different type of locking situation. Any 3 patterns would not have determined a locking situation (within a comparable tolerance) of the type in fig. 2.

#### **Analysis of the parameters characterising a locking situation.**

Some of the discussion regarding the characterisation of the locking situation involved the angles of the triangle determined by the patterns. At the same time, the definition of the locking situation used the slopes of the extreme hyperplanes. Let us analyse the relationship between the slopes of the hyperplanes and the angles of the triangle determined by the patterns (fig. 7). In all the following pictures, the hyperplanes are assumed to be infinitesimally displaced with respect to the patterns so that they classify correctly the patterns.

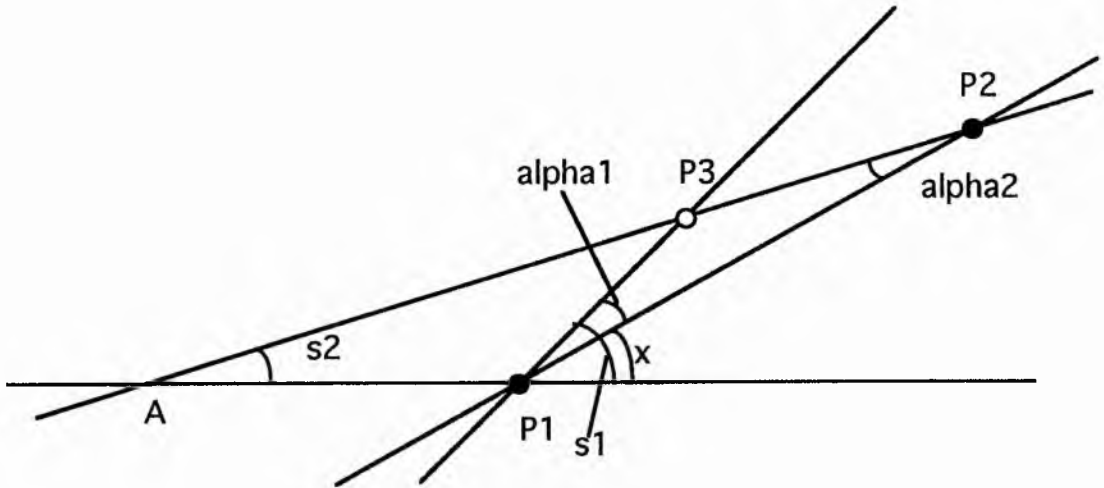


Fig. 7 The relationship between the slopes of the extreme hyperplanes and the angles of the triangle determined by the patterns.

The slope of  $P_1P_3$  is determined by the angle formed by  $P_1P_3$  with the horizontal  $AP_1$  which can be written as:

$$\alpha_1 = s_1 - x \quad (1)$$

From triangle  $AP_1P_2$ , the external angle  $x$  can be written as:

$$x = s_2 + \alpha_2 \quad (2)$$

which can be substituted in (1) to give:

$$\alpha_1 = s_1 - s_2 - \alpha_2 \text{ or}$$

$$s_1 - s_2 = \alpha_1 + \alpha_2 \quad (3)$$

This shows that the difference in slopes is equal to the sum of the angles of the pattern triangle and proves the following lemma.

**Lemma.**

Let  $p_1, p_2$  be two patterns from class  $C_1$  and  $p_3$  a pattern from  $C_2$ . The patterns  $p_1, p_2$  and  $p_3$  determine a locking with tolerance  $\epsilon$  if and only if the sum of the angles corresponding to the patterns  $p_1$  and  $p_2$  in the triangle  $p_1, p_2, p_3$  is less than  $\epsilon$ .

The slopes of the hyperplanes were used in the definition of the locking situation because they offer a higher degree of generality. A definition using the slopes of the hyperplane can be extrapolated to higher dimensions easier than a definition which would use the angles of the triangles. The angles of the triangle are more convenient to use as a criterion because they are independent of the position of the patterns with respect to the axes. This lemma makes the connection between the definition of the locking which uses the slopes of the hyperplanes and the empirical observation regarding the angles of the triangle. The empirical statement that the locking is present when the pattern triangle is squashed ( $\alpha_1 + \alpha_2$  is small) is now supported. Furthermore, this lemma allows us to analyse a locking situation independently of the pattern's position with respect to the axes (i.e. for any orientation of the triangle).

Now, let us analyse the influence of various parameters using the more precise characterisation of the locking situation given by the definition. As suggested by the intuitive analysis presented before, these parameters are the distances  $d$  and  $x$  as shown in fig. 8

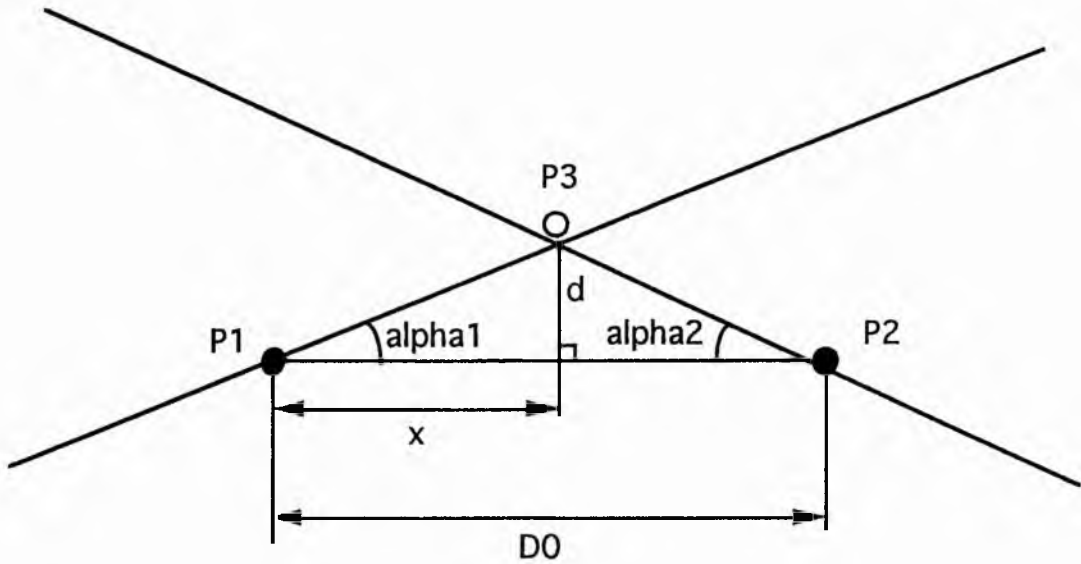


Fig. 8 Some parameters characterising this type of locking situation

The locking with a tolerance  $\epsilon$  is defined by:

$$\alpha = \alpha_1 + \alpha_2 = \arctan \frac{d}{x} + \arctan \frac{d}{D_0 - x} \quad (4)$$

When  $x$  decreases:

$$\lim_{x \rightarrow 0} \alpha = \lim_{x \rightarrow 0} \arctan \frac{d}{x} + \lim_{x \rightarrow 0} \arctan \frac{d}{D_0 - x} = \frac{\pi}{2} + \arctan \frac{d}{D_0} \quad (5)$$

The limit (5) justifies the empirical observation that the tolerance of the locking determined by three patterns as in fig. 1 increases as P3 gets closer to P1 or P2 at a constant distance from the line P1P2. The limit (5) says that if  $x$  is very small, the tolerance of the locking situation is large (approximately  $\pi/2$ ).

When  $d$  increases:

$$\lim_{d \rightarrow \infty} \alpha = \lim_{d \rightarrow \infty} \arctan \frac{d}{x} + \lim_{d \rightarrow \infty} \arctan \frac{d}{D_0 - x} = \pi \quad (6)$$

The limit (6) justifies the empirical observation that the tolerance of this locking situation increases as P3 gets further away from P1 and P2.

### 5.3.1.2 Three dimensions

Let us suppose there are two classes C1 and C2 in a 3 dimensional space. A locking position is presented in fig. 9 In this figure, the white patterns (C1) are on one side of the plane and the black pattern (C2) is on the opposite side. If the distances between the plane and the patterns are very small, the plane is pinpointed in its position. An important aspect of the positions of the patterns in this situation is the fact that the projection of the black pattern on the dividing hyperplane falls in the triangle determined by the projections of the white patterns on the same hyperplane. The extreme positions of the plane so that all patterns are still correctly classified are infinitely close to the sides of the tetrahedron determined by the 4 patterns. Other important elements are the angles formed by faces of the tetrahedron which can be put into correspondence with the gradient of the planes determined by the edges of the tetrahedron along each axis in the same way the angles of the pattern triangle were put into correspondence with the slopes of the lines in the 2D case.

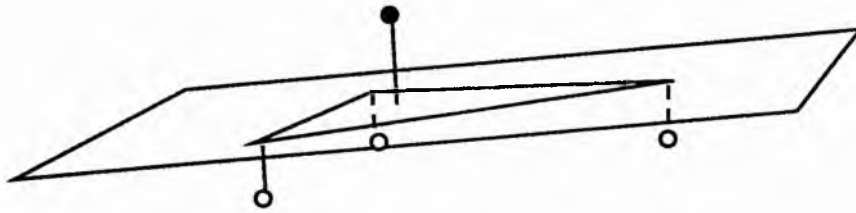


Fig. 9 A locking position in a 3D space.

#### 5.3.1.3 N dimensions

One is now prepared to attempt a generalisation of this particular type of locking situation to  $N$  dimensions. From the 2-D examples above, it is clear that this type of locking situation is characterised by two elements: i) the projection of the black pattern on the dividing hyperplane must fall in the convex hull determined by the projections of the patterns from the opposite class (the projection condition) and ii) the tolerance of the locking is controlled by the orientation of the hyperplanes which form the convex hull<sup>1</sup> determined by the patterns from both classes (the slope condition).

#### A first attempt

Let us suppose there are two classes  $C1$  and  $C2$  in an  $N$  dimensional space. A possibility for pinpointing the position of a hyperplane in an  $N$  dimensional space is to have  $N+1$  points of which  $N$  are on one side of the hyperplane and one is on the other side of it. This would be the  $N$  dimensional generalisation of the locking

---

<sup>1</sup>Strictly speaking, a convex hull is not formed by hyperplanes. The same condition could be formulated as follows: "...orientation of the affine spaces containing the facets of the convex hull determined by ...."

situation presented in fig. 1. Although there are other types of locking situations, the analysis will concentrate on this type.

In order to pinpoint the position of a hyperplane in a locking situation analogous to the 3D one presented in fig. 9, one needs  $N$  points from one class ( $C1$  for instance) and one point for the other class ( $C2$ ). For this situation, the extreme positions of the dividing hyperplane so that all patterns are still correctly classified are determined by all the combination of  $N-1$  patterns from  $C1$  and the pattern from  $C2$ . Each such combination determines a hyperplane and all these hyperplanes determine a simplex in the  $N$  dimensional space<sup>2</sup>. If all these hyperplanes are close (the simplex is squashed towards the dividing hyperplane) then this hyperplane is pinpointed and the locking has occurred.

The locking test can be a comparison of the gradients of the hyperplanes determined by combinations of  $N-1$  points from one class and 1 point from the other class. In other words, the partial derivatives of the surfaces (hyperplanes) determined by such combinations of points with respect to all variables should be close to each other. How close these slopes should be depends on the problem and can be roughly estimated taking into consideration the distance between patterns and the area of interest of the input space. An example of such estimation is given in appendix 2 for the 2D case.

A heuristic for locking detection based on the ideas presented above is given in the following. The algorithm is given for the particular case in which the number of the patterns in one class is equal to the dimensionality of the space  $N$  and the other class contains just one pattern. The justification for this will be given just few paragraphs later.

---

<sup>2</sup> A simplex is the geometrical figure consisting, in  $N$  dimensions, of  $N+1$  points (or vertices) and all their interconnecting line segments, polygonal faces, etc. In two dimensions, a simplex is a triangle. In three dimensions it is a tetrahedron, not necessarily the regular tetrahedron.



**Heuristic 1 for locking detection (in an N dimensional space)**

**detect locking** with points  $p_1, p_2, \dots, p_N$  from C1 and 1 point  $p_{N+1}$  from C2 **is:**

**if** the projection condition is not satisfied **then**

locking has not been detected

**for** each  $i$  from 1 to  $N$  **do**

leave  $p_i$  out and build a hyperplane with the rest of the points (thus, all the combinations of  $N-1$  points from one C1 and 1 point from C2 will be considered)

compare its slopes with the slopes of the hyperplane determined by  $p_1, p_2, \dots, p_N$

**if** all slope differences are smaller than the slope threshold **then**

locking has been detected

**else**

locking has not been detected

Heuristic 1 may not appear to be all that good because it introduces some new problem-dependent parameters. It must be said though, that the ranges of these parameters are not crucial. If the slope error parameter is smaller than necessary, the heuristic will detect only very tight locking positions which means that some time might be wasted in hopeless training sessions. However, the global solution will not be affected and still some locking situations will be detected which will bring a speed improvement. If the slope error parameter is larger than necessary, the heuristic will stop the training in some situations in which the position of the hyperplane could still be adjusted. This could lead to using more hidden units than necessary but a solution will still be found. Furthermore, if other mechanisms for eliminating redundancy are used, the number of the hyperplanes used in the solution can remain small.

A further observation can be made. An exhaustive search for this type of locking would include all the possible combinations of  $N$  patterns from one class and one pattern from the other class. If the number of patterns in each class

(ordinal(C1UC2)) is much larger than the number  $N$  of dimensions of the space which is usually the case, this search could take a long time. However, this exhaustive search is not necessary for the purpose of the search is to investigate the locking of the hyperplane in the current position. Therefore, only those patterns which are close to the hyperplane are probable to lock it. Consequently, only the  $N$  patterns closest to the hyperplane need be taken into consideration when locking is investigated and this is the justification for giving the algorithm only for the particular case involving  $N$  points from one class and one point from the other class.

This heuristic involves considering the  $N$  points closest to the boundary from each class. The number of combinations of one pattern from one class and  $N$  patterns from the opposite class as required by the algorithm given is  $O(N)$ . For each set of points,  $N$  hyperplanes must be compared and each hyperplane comparison takes  $O(N)$  gradient comparisons (one for each dimension). Therefore, the algorithm needs  $O(N^3)$  operations. The number of operations needed for checking the projection condition is  $O(N(N-1)/2+1)+O(N(N-1)/2+1)\log N$  for constructing the convex hull of the  $N$  patterns closest to the dividing hyperplane<sup>3</sup> and  $O(N)$  for checking the projection condition.. Note that  $N$  is the number of dimensions of the input space, not the number of the patterns in the training set and therefore is always less than the number of patterns. If the number of dimensions is larger than the number of patterns, the patterns (assumed to be in general position) are linearly independent and the classes are separable.

As shown, the number of operations could be large, and any reduction of it could be useful. The use of a different heuristic could allow a reduction of the number of operations without too much loss of efficiency.

#### **A less expensive (in terms of computational time) heuristic for locking detection**

The assumption of this heuristic is that the number of patterns is large and if the locking is present it is caused by many patterns from each class.

In these conditions, one could consider the  $N$  points closest to the dividing hyperplane from each class and calculate the hyperplane determined by them. If the

---

<sup>3</sup> ([Preparata, 1985] - Theorem 3.14, pp. 136) where the number of dimensions is  $N-1$

two hyperplanes are close enough (subject to a given error limit) and if the convex hulls determined by their projections on the dividing hyperplane intersect (and the intersection is not degenerate i.e. there is no  $N-2$  dimensional subspace which includes the intersection), then the locking is present.

The  $N$  closest points to the boundary need to be found. Note that the boundary changes at each step and therefore, the  $N$  closest points can change. Finding the  $N$  closest points from a class, needs  $O(M)$  operations where  $M$  is the number of patterns in that class.

The use of this heuristic involves using the  $N$  closest points to the dividing hyperplane to determine a border hyperplane for each class. Then, the real boundary is compared with the two class boundaries. If all three are close to each other the  $N$  closest points from each class are projected on the boundary. If the hulls thus determined intersect, the locking is present. Note that the border hyperplane is none of the extreme hyperplanes (because an extreme hyperplane is infinitesimally close to patterns from both classes whereas a border hyperplane is determined by patterns from one class only).

**Heuristic 2 for locking detection (in an N dimensional space)**

detect locking with points  $p_1, p_2, \dots, p_M$  from  $C_1$ ,  $p_1, p_2, \dots, p_K$  from  $C_2$  and the dividing hyperplane  $d$  is:

find the  $N$  closest points from  $C_1$  to  $d$  and use them to  
construct a border hyperplane  $b_1$  for  $C_1$

find the  $N$  closest points from  $C_2$  to  $d$  and use them to  
construct a border hyperplane  $b_2$  for  $C_2$

if  $b_1$ ,  $b_2$  and  $d$  do not have close gradients i.e.

$$\|grad(b_1) - grad(b_2)\| > \varepsilon \text{ or}$$

$$\|grad(b_1) - grad(d)\| > \varepsilon \text{ or}$$

$$\|grad(d) - grad(b_2)\| > \varepsilon \text{ then}$$

locking has not been detected

stop

project the  $N$  points from  $C_1$  on  $d$  and construct the convex hull  $H_1$  of the projections

project the  $N$  points from  $C_2$  on  $d$  and construct the convex hull  $H_2$  of the projections

if  $H_1$  and  $H_2$  intersect then

locking has been detected

else

locking has not been detected

Two examples of such locking situations in a 2D space are presented in fig. 10.

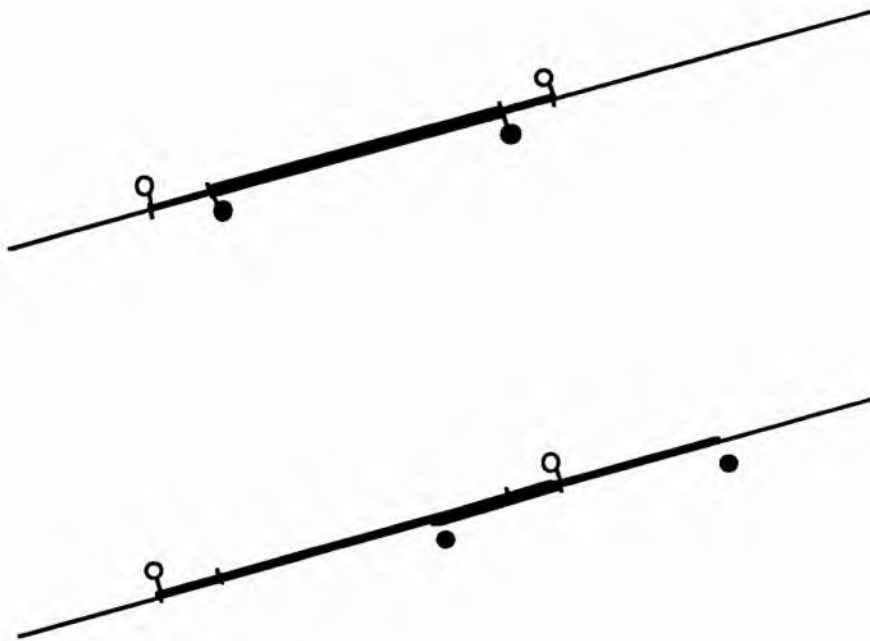


Fig. 10 Two examples of locking positions which will be detected by heuristic 2.

Note that if the hulls do not intersect, the locking is not present. In fig. 11, for instance, the dividing hyperplane can have almost any orientation as long as it intersects the segment determined by patterns 2 and 3. A case in which the intersection is degenerate is presented in fig. 12. If the patterns are assumed to be infinitely close to the dividing hyperplane, a boundary correctly classifying the patterns can be placed in almost any position as long as it contains the intersection point M.

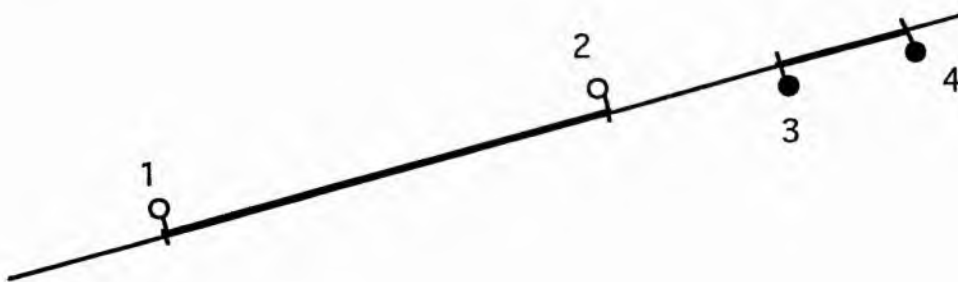


Fig. 11 A non-locking situation. The hyperplanes determined by the  $N$  closest points are close to each other but the convex hulls determined by the projection of the points do not intersect.

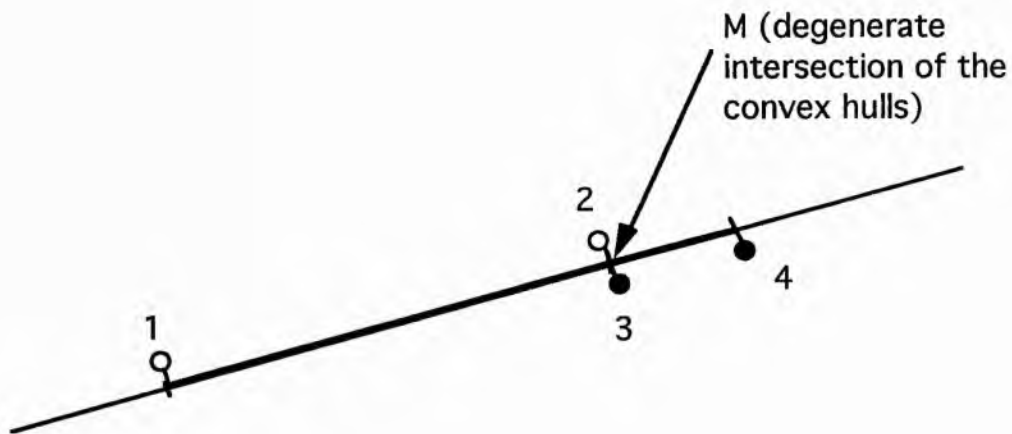


Fig. 12 A non-locking situation. The hyperplanes determined by the  $N$  closest points are close to each other but the intersection of the hulls is degenerated (just one point).

The heuristics presented are expected to succeed in many cases, thus improving the global efficiency of the algorithm but they are not guaranteed to do so. For instance, there are locking situations which will not be detected by heuristic 2 (see fig. 13) because the assumptions made by this heuristics are not true. In fig. 13, the patterns are assumed to be very close to the dividing hyperplane. In this situation, the hyperplane can not be rotated around  $AB$  because of patterns projected in  $C$  and  $D$  and it can not be rotated around  $CD$  because of the patterns projected in  $A$  and  $B$ . The first heuristic (heuristic 1) will construct the hyperplanes determined by all combinations of three points and will detect the locking by comparing their slopes but the second one (heuristic 2) does not have enough patterns from each class to calculate the border hyperplanes and will therefore fail.

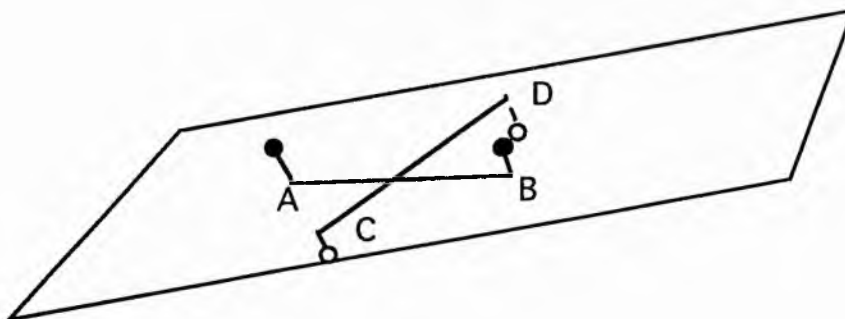


Fig. 13 A locking situation which will be detected by heuristic 1 and will not be detected by heuristic 2.

### 5.3.2 Locking in CBD vs. locking in candidate elimination technique

In [Mitchell, 1978], a candidate elimination approach to rule learning in symbolic artificial intelligence is presented. This approach involves representing and revising the set of all hypotheses describable within the given generalisation language that are consistent with the given training instances. This set of possible generalisations is referred to as the version space of the target generalisation with respect to the given generalisation language.

This approach uses two sets of generalisations  $S$  and  $G$ .  $S$  contains the most specific generalisations that are consistent with the observed training instances and  $G$  contains the most general generalisations that are consistent with the observed training instances. A generalisation  $x$  is contained in the version space represented by  $S$  and  $G$  if and only if  $x$  is more specific than or equal to some member of  $G$  and  $x$  is more general than or equal to some member of  $S$ . The partial ordering relation more-specific-than is crucial to this technique. The training adds individual patterns and modifies  $S$  and  $G$  so that they satisfy their definition.

The training can terminate in two situations: i)  $S$  becomes equal with  $G$  and ii) all the training instances have been considered. If  $S$  becomes equal with  $G$ , the version space contains a single element  $S=G$  which is the only possible generalisation consistent with all training instances. At this moment, the training can be stopped because any other training instance will be either redundant or inconsistent. A training instance is said to be redundant if its adding to the pattern set does not change the current version(s) of the concept to be learnt. A training instance is inconsistent if it is not consistent with  $S$  or  $G$ . The second termination scenario is when all patterns in the training set have been considered and  $S$  is still different from  $G$ . In this situation, there are many possible generalisations and the set of training instances is not sufficient to uniquely select one of them. Any generalisation more general than  $S$  and more specific than  $G$  is a valid generalisation.

The most important improvement brought by the version space approach is the possibility of detecting the moment in which the only possible generalisation has been found. This relies heavily on two crucial elements: the partial ordering and the generalisation language. The partial ordering ensures that if a generalisation  $G_1$  is more specific than another generalisation  $G_2$  and  $G_1$  is requested by the training instances, then all the other possible generalisations in-between  $G_1$  and  $G_2$  with respect to the partial order relation can be eliminated. This ensures the volume of the

version space is reduced each time S or G are modified until eventually, it contains a single generalisation, the concept to be learned. The generalisation language is crucial because it eliminates all the attributes which are not necessary for describing the concept to be learnt and introduces the bias necessary for any concept learning [Mitchell, 1980], [Mitchell, 1982].

A neural implementation of Mitchell's symbolic technique is proposed in [Weir, 1993]. In the neural version, the sets S and G are implemented by two networks which use different error functions. Unfortunately, this approach seems to fail according to the authors because of the many-many relationship between concepts and specificity measures. Thus, this type of relationship prevents defining a partial ordering of neural equivalent of the version space.

A different approach to implementing Mitchell's technique in a neural network framework is discussed in [Weir, 1994]. Two measures believed to be able to induce a partial ordering of the neural equivalent of the version space are proposed. A first measure is the distance between the hyperplanes and the patterns. A second measure is the angle in weight space between the weight vector corresponding to the current position and some reference weight vector. This second measure is investigated in more detail and a limited implementation (one layer networks) is presented. In this approach, the locking of S and G appears when they both form the same angle with the reference vector. Since S was initialised with the reference vector and G was initialised with the opposite of the reference vector, the locking is interpreted to signal a complete exploration of all possible angles. In the interpretation of the authors, this locking in the weight space corresponds to a locking situation like that presented in fig. 1.

Now, the resemblance between the locking in Mitchell's candidate elimination approach and the locking in constraint based decomposition can be illustrated. CBD was not derived as a neural implementation of Mitchell's technique and it is not seen as being such. Having said that, once certain correspondences are established, some analogies and common features can be illustrated. The version space in Mitchell's technique corresponds to the set of all possible positions of the boundary (i.e. the unconstrained area in fig. 2) in constraint based decomposition. In both cases, the convergence is performed (or at least attempted if there aren't enough patterns available) by consecutive reductions of the version space. In the symbolic technique, this is done by updating S and G. In the neural technique, the reduction of the version space is performed by extending the area definitely assigned to each



of the two classes (area C1 and C2 in fig. 2). Initially, the version space is the whole space. The first instance in the symbolic technique initialises S and G so that the first instance is correctly classified and the version space is still as large as possible. The same is done in the neural technique. Locking occurs when the version space is reduced to a singleton in the symbolic technique. In the neural technique, the locking occurs when the version space becomes so small that the differences between different possible generalisations are below the error limit and therefore, their different results can be identified.

#### **5.3.2.1 Inconsistencies in the training set.**

In its original form, the version space approach cannot cope with inconsistent sets of training instances. If the training set is inconsistent, the algorithm will modify S and G up to a certain point when an inconsistency is detected. This will be signalled and the algorithm will stop.

There are at least two different types of inconsistencies. A possible inconsistency can be generated by two equal instances, one classified as a positive instance and the other classified as a negative one. In the case of a perceptron used to separate two classes, this would correspond to the presence of the same pattern in the training set of both classes. In this case, one of the instances (or both) must be eliminated from the training set.

The second type of inconsistency occurs in the version space technique when a training instance is not consistent with S or G but does not contradict directly any previously considered instance. This shows a failure of the generalisation language to embody all the significant features of the concept to be learned. In other words, the generalisation language is not powerful enough to define correctly the concept. In the neural case, this would correspond to the situation in which a new pattern makes the problem linearly inseparable. Although the new pattern does not contradict directly any other training pattern (by assigning the same input values to a different class), the perceptron will not be able to find a solution. Even in this case, the model is not powerful enough to solve the problem.

#### **5.4. Improving the training speed through avoiding problems beyond the possibilities of the current architecture**

The efficiency of the global algorithm can be further improved. The locking detection ensures that the training of a given neuron will not be continued if the

position of the hyperplane implemented by the given neuron cannot be changed anymore. However, there are situations in which a particular problem cannot be solved even if the position of the hyperplane has not been fixed yet. The constraint based decomposition algorithm works by adding patterns one by one and trying to improve the position of the hyperplane so that even the new patterns are correctly classified. There are situations in which the hyperplane can be moved, i.e. it is not locked, but the position of the last added pattern is such that the problem is not linearly separable. In this situation, the algorithm will hopelessly try to modify the position of the hyperplane and will eventually detect the failure when the time-out condition becomes true. This can take a long time. Some heuristic or algorithm able to eliminate those problem which are not linearly separable would save a lot of training time.

This heuristic should take into consideration the current position of the hyperplane, the current training set (a number of patterns from each class) and the new pattern and should decide whether the problem obtained by adding the new pattern is still linearly separable.

#### **5.4.1 Linear separability**

**Definition.** Two sets are said to be linearly separable if and only if there exists a hyperplane  $H$  that separates them.

**Problem 1.** Given a set of patterns  $p_1, p_2, \dots, p_n$  from two classes  $C_1$  and  $C_2$ , decide if the set is linearly separable.

**Theorem 1<sup>4</sup>** Two sets of points are linearly separable if and only if their convex hulls do not intersect.

The theorem tells us that for the present purpose of establishing linear separability, convex hulls, rather than say concave hulls, should be the focus of the problem.

Using this criterion, problem 1 can be solved by constructing the convex hulls of the points in each class. The construction of the intersection can then be attempted. If the intersection is non-empty, the classes are not linearly separable.

---

<sup>4</sup>[Preparata, 1985] - Theorem 7.1, pp.. 269

This is a standard approach to the problem of linear separability. There are several reasons for which this approach is not well suited to the problems raised by a constructive training algorithm for a neural network. Firstly, the standard approach sees the construction of the convex hull and the test for their intersection as two different problems and treats them sequentially. The convex hulls are constructed first, and only then their intersection is constructed (or detected). The approach presented in this thesis merges the two phases which can be more efficient in many cases. The efficiency comes from the fact that: i) one does not have to keep the whole convex hull of the patterns but only a minimal hull and ii) when a new point is added, the hull does not have to be updated always and iii) if the problem is linearly inseparable, the algorithm can detect this without necessarily using all patterns.

In the following sections, an algorithm oriented towards the requirements of the neural network training and which merges the two phases of the linear separability test will be presented.

#### **5.4.1.1 Two dimensions.**

In 2D, the construction of the convex hulls of  $n$  points can be performed in  $O(n \log(n))$  time using  $O(n)$  space which is optimal<sup>5</sup>. The intersection of two convex polygons with  $n$  and  $m$  vertices can be found in  $\theta(n+m)$  time<sup>6</sup>. Therefore, answering the problem by using this approach needs  $O(n \log(n)) + O(m \log(m)) + O(m+n) = O(n \log(n))$  operations where  $m$  and  $n$  are the number of vertices of the convex hull of each class.

If an on-line algorithm is used for the construction of the convex hull, the update time is<sup>7</sup>  $\theta(\log(n))$  (the total time is<sup>8</sup>  $\theta(n \log(n))$  for the construction of the convex

---

<sup>5</sup>[Preparata, 1985] - Theorem 3.7, pp.. 109

<sup>6</sup> [Preparata, 1985] - Theorem 7.3, pp.. 272

<sup>7</sup>[Preparata, 1985] - Theorem 3.11 pp.. 123

<sup>8</sup>[Preparata, 1985] - Theorem 3.11 pp.. 123

hulls only. The global update time becomes  $\theta(\log(n)) + O(m+n) = O(m+n)$  where  $m$  and  $n$  are the numbers of vertices of each hull respectively<sup>9</sup>.

An example of a 2D set of patterns from two classes which are linearly separable is given in fig. 14

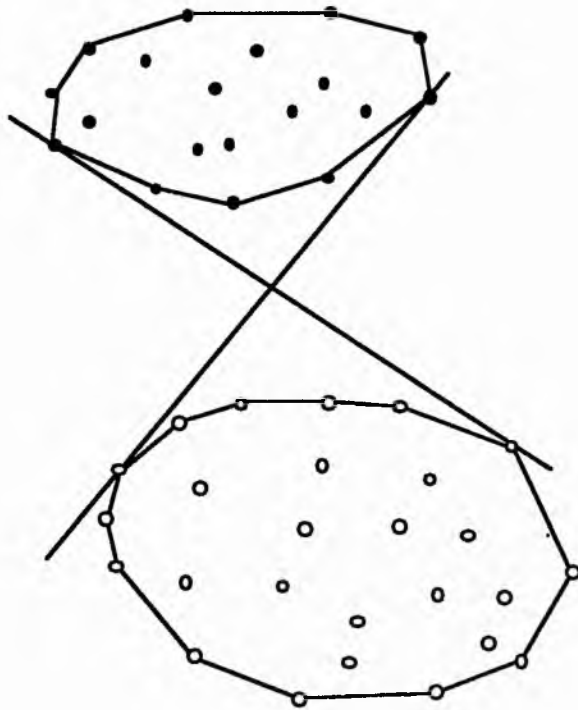


Fig. 14 A linearly separable set in 2D. The two supporting lines are the extreme possible positions of the separating line.

Problem 1 can be reformulated as problem 2.

**Problem 2.** Given a set of patterns  $p_1, p_2, \dots, p_n$  from two classes  $C_1$  and  $C_2$ , a hyperplane  $H$  which separates them, and a new pattern  $p_m$  from either  $C_1$  or  $C_2$ , decide if the new problem  $p_1, p_2, \dots, p_n, p_m$  is linearly separable. If the problem is linearly separable, find a new position of the separating hyperplane so that all patterns ( $p_m$  included) are correctly classified.

---

<sup>9</sup> $f$  is  $\theta(g(n))$  if there exist  $c_1, c_2$  and  $n_0$  such that  $c_1 g(n) \leq f(n) \leq c_2 g(n)$  for any  $n \geq n_0$ ;  $f$  is  $O(n)$  if there exist  $c_1$  and  $n_0$  such that  $f(n) \leq c_2 g(n)$  for any  $n \geq n_0$ .

Problem 2 is exactly the problem faced by the CBD constructive algorithm each time a new pattern is added.

It is easy to see that if an algorithm for problem 2 existed, problem 1 could be solved by starting with 2 points from different classes, adding one pattern at a time and sequentially using the algorithm for problem 2. An algorithm for problem 2 can be seen as an on-line version of the algorithm for problem 1 with the property that it builds the solution.

As the neural application requires finding the position of the separating hyperplane, a constructing algorithm is better than an algorithm which simply tells whether the classes are linearly separable or not.

**Theorem.**(2 Dimensions)

For any linearly separable set, there exist two separating lines which are supporting lines for the convex hulls of the classes with the property that any other separating line will have the slope in-between the slopes of these two supporting lines.

**Proof.** (by construction).

As the patterns are linearly separable, there exists a line which separates them. Let  $d$  be a separating line.

Find the patterns from each class which are closest to the separating line  $d$  (see fig. 15). Let these patterns be  $p_1$  and  $p_2$ . Build the segment  $p_1p_2$  and find its intersection  $O$  with the separating line  $d$ . The segment  $p_1p_2$  exists because  $p_1$  and  $p_2$  are patterns from different classes. The intersection between the separating line and the segment exists because the classes are separated and therefore patterns from different classes are in different half-spaces with respect to  $d$ .

Rotate  $d$  anticlockwise around  $O$  until a pattern  $P_1$  of  $C_1$  (be it  $C_1$ ) is found. In this position, the line  $d$  is a supporting line for the convex hull of class 1. Alternatively, this position can be found by comparing the polar angles determined by  $d$  and  $Op$  for any  $p$  in class 1 and choosing the pattern with the smallest polar angle.

Rotate anticlockwise the line around  $P_1$ , until a pattern from the opposite class is found (see fig. 16). In this position, line  $d$  is a supporting line for both convex hulls.

In order to construct the other dividing and supporting line, start from  $O$  using rotation in the opposite sense.

The resulting supporting lines are as those in fig. 14 and from the construction method it follows that any other separating line will have a slope in-between the slopes of the two supporting lines. This is because, one of the resulting supporting lines is obtained by starting from an initial position and increasing the slope whereas the other is obtained by decreasing the slope.

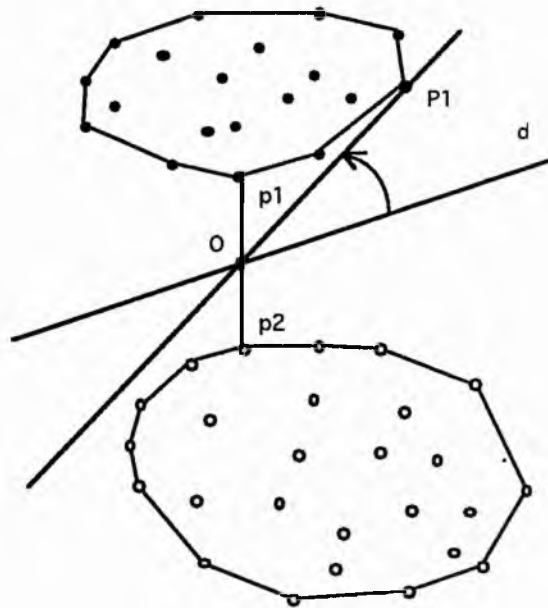


Fig. 15 Finding an extreme position of the separating line. Find the pattern in each class which is closest to the separating line ( $p_1$  and  $p_2$ ). Find the intersection of  $p_1p_2$  with  $d$  (point  $O$ ). Rotate  $d$  around  $O$  until it reaches a pattern (or several patterns in the degenerate case in which a facet of the convex hull contains more than 2 patterns)

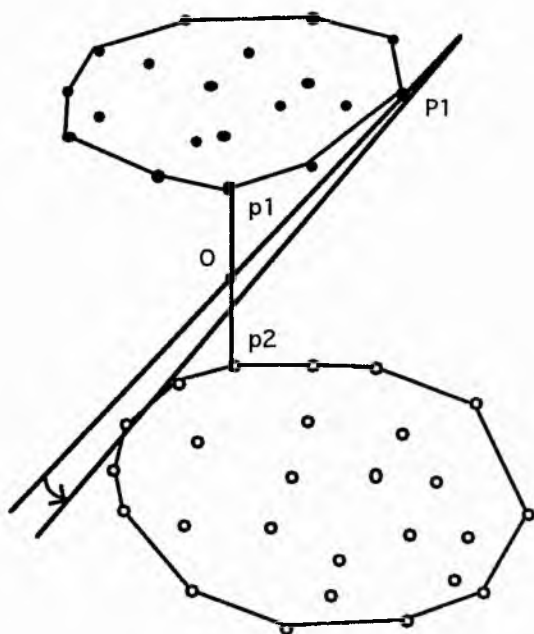


Fig. 16. The line is then rotated in the opposite sense around P1.

The fact that the two supporting lines are the extreme positions of a separating line can be used to solve problem 2 more efficiently. Instead of considering the convex hulls of each class, one can consider the input space divided into different areas as in fig. 17. Any black pattern in area 2 or any white pattern in area 1 will make the problem linearly inseparable. The problem will remain linearly separable if a pattern from any class is situated anywhere outside areas 1 and 2. Area 1 will be called the shadow of class 1 (where class 1 is the one which contains black patterns) and area 2 will be called the shadow of class 2 (class 2 contains white patterns). A light source placed in the intersection of the extreme supporting lines would have areas 1 and 2 in the shadow of the two convex hulls.

The shadow areas can be determined taking into consideration only a part of the pattern set in each class as shown in fig. 18. and this is one factor which can improve the expected efficiency of the approach (as opposed to the worst case one).

The convex hull that contains the least number of vertices from the set of vertices of the complete hull and determines the same shadow will be called the minimal convex hull.

A facet of the convex hull of a class will be called a front facet if its affine space (a line for the 2D case) separates the classes.

Let us consider the patterns which belong to exactly one front facet. We shall call these patterns corner patterns. For the 2 dimensional case there are two such patterns for each hull. These patterns belong to the extreme hyperplanes as well. The minimal hulls can be obtained by considering all the front facets of the hulls and the facet determined by the two corner patterns of each hull.

The minimal convex hull has the property that it has at most one common facet (edge) with the shadow area (see fig. 18).

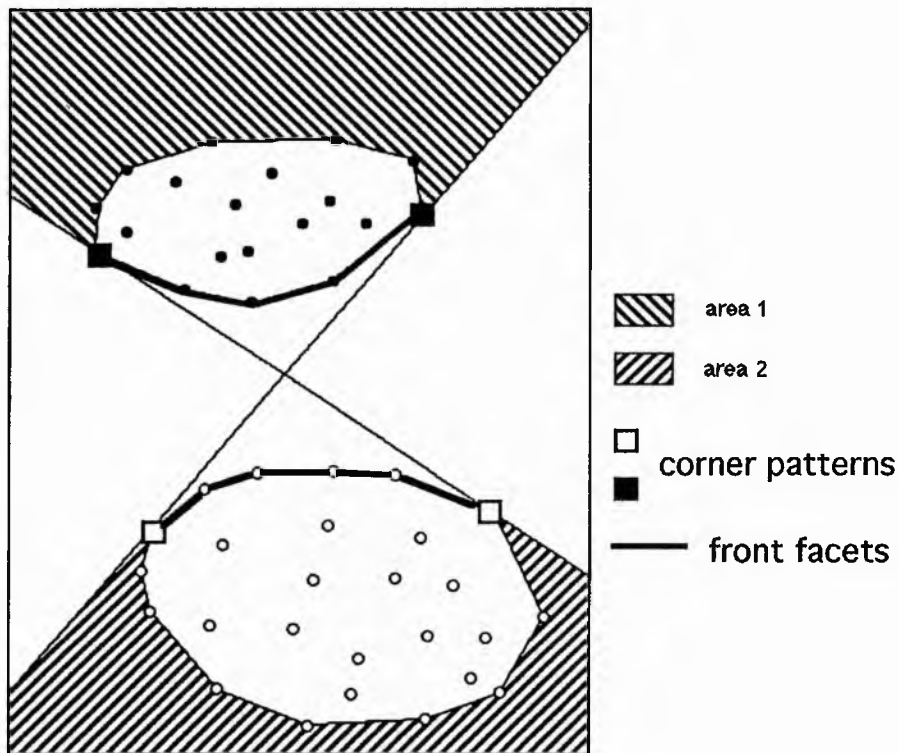


Fig. 17 Shadow cones determined by the convex hulls. Any new black pattern in area 2 or any new white pattern in area 1 will transform the problem in a linearly inseparable one.



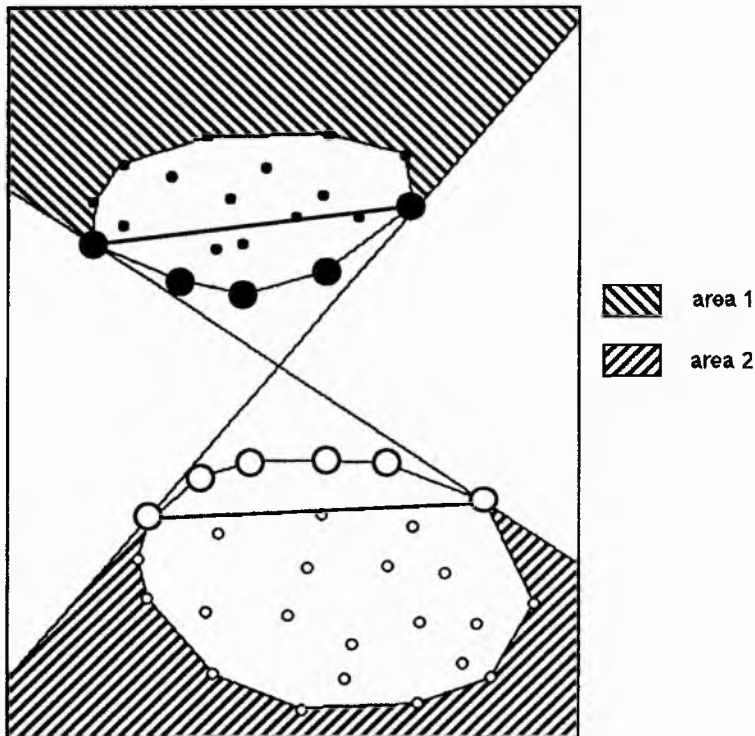


Fig. 18 Not all patterns need be considered for determining the shadow area 1 and 2. A set containing only the big patterns will determine the same shadow areas as the original set.

**Algorithm for deciding on-line the linear separability of a set of patterns from two classes in 2D.** The algorithm produces a separating line as well.

The algorithm starts with a minimal number of patterns (one from each class in order to ensure the initial linear separability) and builds and maintains the minimal convex hulls of each class. When a new pattern is added, the minimal convex hulls are used to determine if the new problem is linearly separable and if the separating line needs updating.

For describing the algorithm, let us consider the situation in fig. 19. A certain number of patterns from each class form the minimal hulls of the classes.

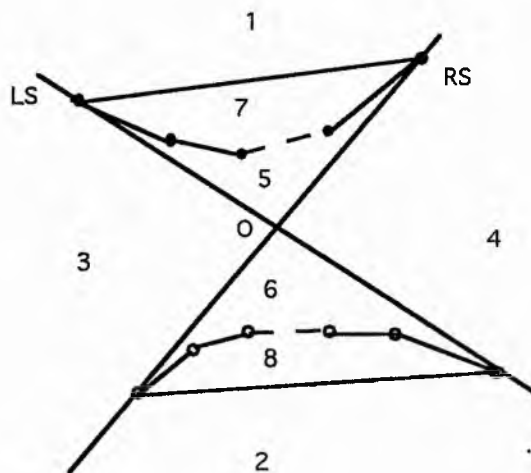


Fig. 19. The current situation to be considered by the algorithm.

Let us suppose a new black pattern is added. If the new pattern is situated in area 1 or 7 (in fig. 19), the new problem is linearly separable and the separating line can remain the same. The shapes of the shadow areas and of the minimal hulls remain also the same and the new pattern can be removed from further consideration.

If the new black pattern is situated anywhere in areas 3,4,5 or 6, the minimal hull will need updating whereas the separating line might or might not need updating. However, the problem is still linearly separable and a solution will always exist.

The updating of the supporting lines LS and RS depends on the position of the new pattern as well. If the black pattern is situated in area 3, only the left supporting line LS will need updating. If the pattern is in area 4, only the right supporting line RS will need updating. If the pattern is in area 5, neither supporting lines will need updating and finally, if the pattern is in area 6, both of them will need updating (fig. 20).

If the new black pattern is in areas 8 or 2 (in the minimal hull of the opposite class or in its shadow), the problem becomes linearly inseparable and the algorithm terminates.

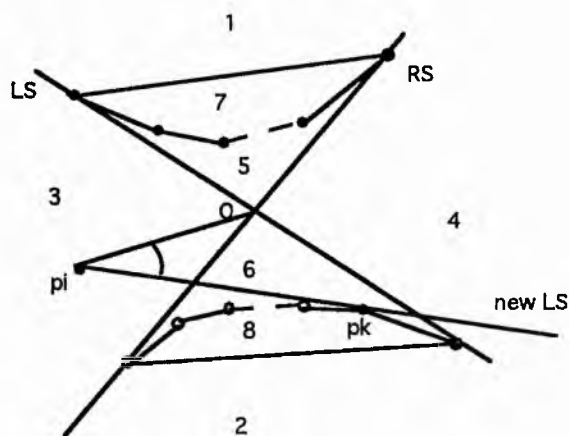


Fig. 20 Updating the left supporting line LS when the black pattern is in region 3 or 6. A pattern  $p_k$  from class 2 (white) will be found so that the  $\angle pip_k$  angle is minimum. The line  $pip_k$  will be the new LS. Both minimal convex hulls need updating.

The total update time is  $O(\log(N_{\max}))$  in the worst case where  $N_{\max}$  is the maximum of  $m$  and  $n$  which are the number of vertices of the minimal hulls (equal to the number of patterns only in the worst case). In general, the number of patterns is greater than the number of patterns in the convex hull of the class which in turn is greater than the number of patterns in the minimal hull. Furthermore, at each iteration there are cases in which only a few operations are needed and the regions corresponding to these cases are quite large (see area 1 in fig. 19). This justifies hopes that the algorithm will perform on average much better than in the worst case.

Thus, in the worst case, the algorithm does not perform worse than an algorithm which constructs the convex hulls<sup>10</sup>, while the expected performance is much better. At the same time, it eliminates the step involving the calculation of the intersection of the convex hulls and builds and maintains the solution (the separating line).

<sup>10</sup>[Preparata, 1985] - Theorem 3.10 pp. 113

### **Algorithm for deciding the linear separability in 2D**

#### **Storage:**

n ordered extreme points of the minimal hull of class 1 (black in fig. 19)  
m ordered extreme points of the minimal hull of class 2 ( $m+n \geq 3$ )  
the left supporting line (LS)  
the right supporting line (RS)  
their intersection O

#### **Algorithm:**

Point  $p_i$  (black) is added.

1. Locate  $p_i$ 
  - 1.1 **if**  $p_i$  is in 2 or 8 **then**
    - 1.1.1 The problem is not linearly separable. Stop.
  - 1.2 **if**  $p_i$  is not in 1 **then**
    - 1.2.1 **if**  $p_i$  is not in 7 **then**  $O(\log(n))$  where  $n$  is the no. of vertices
      - 1.2.1.1 Locate  $p_i$  in 3,4,5 or 6
    - else stop** - the problem is separable and no update is needed
  - else stop** - the problem is separable and no update is needed
2. Update LS
  - 2.1 **if**  $p_i$  is in 3 or 6 **then**
    - 2.1.1 find  $p_k$  in class 2 so that  $Op_k$  is minim
3. Update RS
  - 3.1 **if**  $p_i$  is in 4 or 6 **then**
    - 3.1.1 find  $p_k$  in class 2 so that  $Op_k$  is minim
4. Update convex boundary of class 1
  - 4.1 **if**  $p_i$  is in 3,4 or 5 **then**
    - 4.1.1 update minimal convex hull of class 1
  - else if**  $p_i$  is in 6 **then**
    - minimal convex hull of class 1 degenerates to  $p_i$

The most expensive steps are: to decide if a point is in a convex hull ( step 1.1 of the algorithm), find the pattern which forms the minimum angle with a given line (steps 2.1.1 and 3.1.1) and update a minimal convex hull (step 4.1.1).

The number of operations needed is  $O(\log(m))$  where  $m$  is the number of vertices of the convex hull for step (1.1),  $O(\log(n))$ <sup>11</sup> for (2.1.1) and  $O(\log(m))$  for (3.1.1) and finally  $\theta(\log(n))$ <sup>12</sup> for (4.1.1).

#### 5.4.1.2 N dimensions.

The reformulation of Problem 2 for the  $n$  dimensional case is:

**Problem 2.** Given a set of patterns  $p_1, p_2, \dots, p_n$  from two classes  $C_1$  and  $C_2$ , a hyperplane  $H$  which separates them and a new pattern  $p_m$  from either  $C_1$  or  $C_2$ , decide if the new problem  $p_1, p_2, \dots, p_n, p_m$  is linearly separable. If the problem is linearly separable, find a new position of the separating hyperplane so that all patterns ( $p_m$  included) are correctly classified.

An extension of the 2D algorithm will be presented now.

From the definition of the extreme hyperplanes in 2D one can generalise to  $n$ -D. Thus, in 2D, an extreme hyperplane was a supporting hyperplane for the convex hulls of both classes and which separates the classes at the same time. In the non-degenerate case, such a hyperplane will contain one pattern from each class (see fig. 14). In a multi-dimensional space ( $d$  dimensions) a hyperplane is determined by  $d$  points. Two necessary and sufficient conditions for an extreme hyperplane are: i) to be a supporting hyperplane for the convex hulls of both classes and ii) to separate the classes.

The 2D example presented in fig. 19 will be used to illustrate the algorithm. Although the example is in 2D, the algorithm does not rely on anything specific to this number of dimensions and can be applied in  $n$ -D as well.

The algorithm keeps a list of extreme hyperplane positions for each class. For class 1, each such extreme hyperplane  $h_{s1}^i$  (the  $i$ -th such hyperplane for class 1) is determined by  $d-1$  patterns from class 1 and one pattern from class 2 and is such that the classes are separated with the exception of the  $d$  patterns which belong to the hyperplane. Furthermore, these hyperplanes are such that all the patterns from class 1 are in the positive half-space ( $\geq 0$ ) and all the patterns from class 2 are in

---

<sup>11</sup>([Preparata, 1985] - Theorem 3.11, pp. 123)

<sup>12</sup>([Preparata, 1985] - Theorem 3.11, pp. 123)

the negative half-space. For class 2, each hyperplane  $h_{s2}^i$  will be determined by  $d-1$  patterns from class 2 and one pattern from class 1 and will have the same property of separating the classes with the exception of  $d$  patterns situated on the hyperplane itself.

The algorithm keeps a list  $R_1$  with the points from class 1 used in constructing the hyperplanes  $h_{s1}$  and a list  $R_2$  with the points from class 2 used in constructing the hyperplanes  $h_{s2}$ .  $R_i$  will be used to designate either  $R_1$  or  $R_2$ .

The algorithm keeps the minimal convex hull for each class as a list of facets (and a list of affine spaces - hyperplanes - determined by facets). The hyperplanes are defined so that the interior of the hull is in the positive half-space of each of them. The facets are divided into front facets and back facets. A front facet is a facet whose affine space separates the classes. A back facet is a facet which is not a front facet.

Let us consider the convex hull of each class. The affine spaces determined by facets are hyperplanes. The minimal hull will have all the facets of the convex hull for which the affine space determined by the facet separates the classes and the facets determined by the points in  $R_i$ . As an observation, all the points in  $R_i$  are contained in at least one front facet. The back facets contain exclusively points from  $R_i$ . A hull contains back facets only if it is not degenerate. The hull is degenerated when it is contained in a  $n-1$  subspace of the  $n$  dimensional space and if the hull is degenerated it can contain only front facets.

Let us suppose the next pattern is from class 2. As in the 2D case (see fig. 19), if the pattern is in the convex hull of class 1 or in its shadow, the classes are not linearly separable. If the pattern is anywhere else, the problem is linearly separable. The difference from the 2D situation is that the shadow is not defined by only two hyperplanes (lines in the 2D case) but by many hyperplanes.

Let us suppose  $H$  is the set of hyperplanes  $\{h_{s1}^i\}$  defined by  $d-1$  patterns from class 1 and one pattern from class 2 and such that the classes are separated with the exception of the  $d$  patterns which belong to the hyperplane. Let us suppose a new pattern  $p$  from class 2 is added.

If there exists a supporting hyperplane  $h_{s1}^k$  in  $H$  so that pattern  $p$  is on its negative side, pattern  $p$  is on the opposite side with respect to all the patterns in class 1 (because  $h_{s1}^k$  was defined so that the convex hull of class 1 was on its positive

side). In this case, the classes are linearly separable. This situation is presented in fig. 20b.

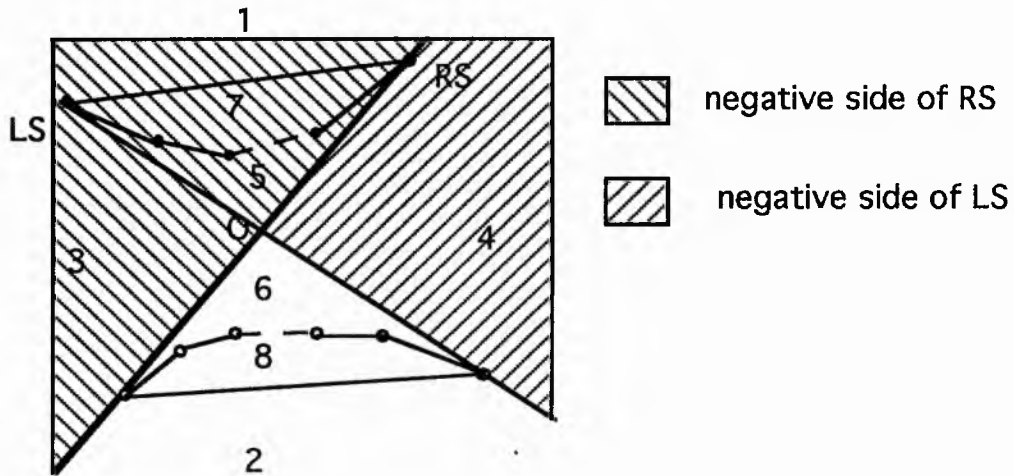


Fig. 20b The negative sides of the supporting lines of the hulls are permissible zones for a pattern from class 2 (black).

If there exists an affine space corresponding to a front facet of the hull of class 1 so that pattern  $p$  is on its negative side, then again  $p$  is on the opposite side of that hyperplane with respect to the patterns in class 1. This is because the sign of the hyperplane (the affine space of the facet of the hull of class 1) was chosen so that the interior of the hull and therefore all other patterns from  $C_1$  were on its positive side. In this situation, the classes are linearly separable as well. This situation is presented in fig. 20c.

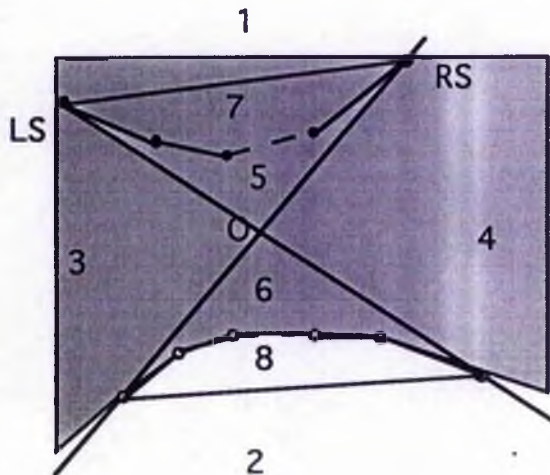


Fig. 20c The union of the negative sides of all the affine spaces of the front facets of the hull of class 1. A pattern from class 2 situated in this area can be separated.

If there is no supporting hyperplane  $h_{s1}^k$  in  $H$  or affine space of a facet so that the new pattern  $p$  is in their negative half-space, then  $p$  is either in the convex hull of class 1 or in its shadow (areas 8 or 2 in fig. 20c). Since  $p$  is from class 2, the problem is not linearly separable anymore.

In conclusion, if there exists a supporting hyperplane  $h_{s1}^k$  in  $H$  so that pattern  $p$  is on its negative side or there exists an affine space corresponding to a front facet of the hull of class 1 so that pattern  $p$  is on its negative side, the classes are linearly separable.

Otherwise, pattern  $p$  is in the convex hull of  $C2$  or in its shadow and the problem is not linearly separable.

If the classes are linearly separable, update the hull of  $C2$  as in the beneath-beyond method<sup>13</sup>. If the pattern is in the shadow of  $C2$  remove it from further consideration. Update those hyperplanes  $h_{s1}^i$  which classify the pattern  $p$  in their positive half-space.

Choose any of the hyperplanes  $h_{s1}^i$  as the updated separating hyperplane.

---

<sup>13</sup>See for instance [Preparata, 1985], pp. 136-140. The beneath-beyond method has been chosen because it can be adapted to on-line functioning. The method is too complicated to be described here. Any other on-line method for updating a convex hull can be used.



Care must be taken in dealing with the degenerate cases, and with the initial situation, before the hulls reach their full dimensionality.

The study of the complexity of this algorithm is not a simple matter. The main obstacle is the fact that the number of operations executed by the algorithm cannot be put into a simple correspondence with the number of points or the number of dimensions of the input space. A relation can be established between the number of facets of the hulls and the number of operations executed by the algorithm but this is not very informative in general because it depends very much on the particular problem. However, one must note that the complexity of the linear separability algorithm is not crucial for the tractability of the problem. The CBD algorithm itself has guaranteed convergence and its number of iterative calls is (at worst) linear with the number of patterns. One could simply monitor the activity of the linear separability mechanism and switch it off if it becomes too time consuming.

Observation. Eventually, any hyperplane with partial derivatives in-between the extreme position hyperplanes can be a valid boundary for the two classes. (A minimum and a maximum can be calculated for the partial derivative of these hyperplanes along each axis. The least maximum and the largest minimum are the boundaries of the valid values of the partial derivatives along each axis. Any hyperplane with the partial derivatives along each axis in-between these values is a valid boundary.)

### Algorithm for deciding (on-line) linear separability

Storage:

$n_1$  hyperplanes  $h_{s1}^i$ . Each  $h_{s1}^i$  contains  $d-1$  points (patterns) from  $C1$  and one point from  $C2$  and separates the classes with the exception of the points contained by  $h_{s1}^i$  itself. The hyperplanes are such that all the patterns from  $C1$  are classified in its positive half-space ( $\geq 0$ ).

$m_1$  hyperplanes which are the affine spaces of the front facets of the convex hull of  $C1$ . Each of them separates the classes with the exception of the points contained by themselves. The hyperplanes are such that all the patterns from  $C1$  are classified in its positive half-space ( $\geq 0$ ).

$n_2$  hyperplanes  $h_{s2}^i$ . Each  $h_{s2}^i$  contains  $d-1$  points (patterns) from  $C2$  and one point from  $C1$  and separates the classes with the exception of the points contained by  $h_{s2}^i$  itself. The hyperplanes are such that all the patterns from  $C2$  are classified in its positive half-space ( $\geq 0$ ).

$m_2$  hyperplanes which are the affine spaces of the front facets of the convex hull of  $C2$ . Each of them separates the classes with the exception of the points contained by themselves. The hyperplanes are such that all the patterns from  $C2$  are classified in its positive half-space ( $\geq 0$ ).

point  $p_i$  from  $C2$  is added.

for each hyperplane of  $n_1$  and  $m_1$

calculate the classification of  $p_i$

if all hyperplanes classify  $p_i$  in their positive half-space then

the problem is not linearly separable

return

/\* update the hyperplanes for  $C2$  \*/

for each hyperplane of  $n_2$  and  $m_2$

calculate the classification of  $p_i$

if all hyperplanes classify  $p_i$  in their positive half-space then

remove  $p_i$  from further consideration because it is in the shadow of  $C2$

(it does not influence the supporting hyperplanes of either class)

update those hyperplanes of  $n_2$  which classify  $p_i$  in their negative half-spaces

update those facets of the minimal hull whose affine spaces classify  $p_i$  in their negative half-spaces (update time is  $O(\phi_{d-1})$  where  $\phi_{d-1}$  is the number of (front) facets<sup>14</sup>).

<sup>14</sup>.[Preparata, 1985] - Theorem 3.16 re. the beneath-beyond method, pp. 140

### **5.4.2 Conclusion**

Unfortunately, the number of the hyperplanes which determine the shadow of a hull with respect to the other cannot be put into correspondence to the dimensionality of the input space or to some properties of the convex hulls themselves. For this reason, for some problems, this algorithm could become costly in terms of computational time to the extent that its use is not justified anymore. However, potentially, this n-D version of the algorithm is able to reduce the training time and can become important when the training is very expensive.

## **5.5. Efficient use of hyperplanes**

### **5.5.1 The problem of redundant hyperplanes**

Due to its divide and conquer approach and the lack of interaction between solving different problems in different areas of the input space, the solution built by the CBD algorithm can be inefficient in some situations in the sense that the solution will not use the minimum number of hyperplanes.

For instance, for the problem presented in fig. 21 with one of its possible optimal solutions there are many non-optimal solutions. One of these non-optimal solutions is shown in fig. 22.

The CBD constructive algorithm could construct non-optimal solution as that in fig. 22, in the following way. The first hidden unit can implement either hyperplane 1 or hyperplane 2. Let us suppose it implements hyperplane 1. Subsequently, the search will be performed in each of the half-spaces determined by the hyperplane 1. At least one hyperplane will be required to solve the problem in each half-space even though one single hyperplane could separate all patterns.

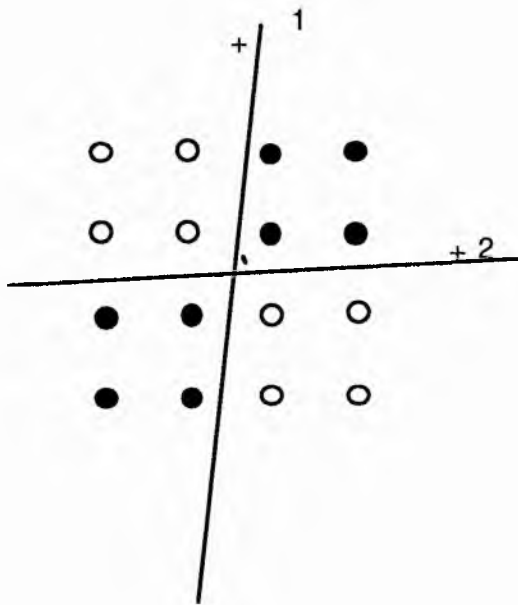


Fig. 21. A problem and a possible minimal solution.

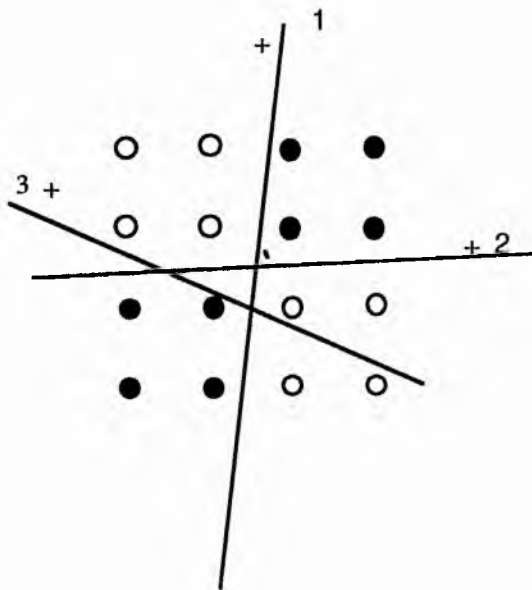


Fig. 22 A solution with 3 hyperplanes.

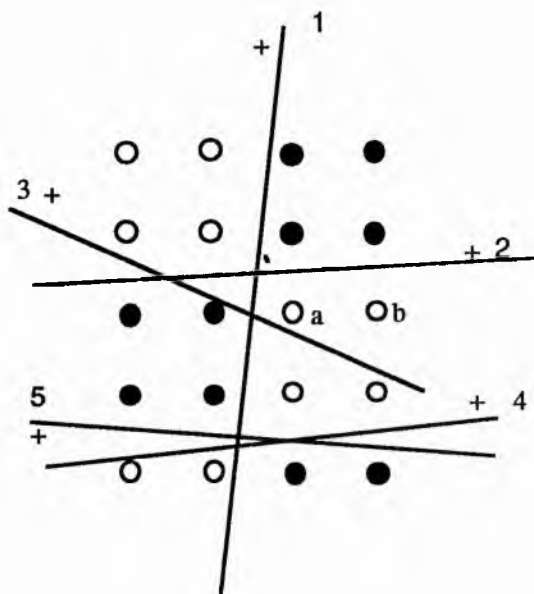


Fig. 23. A more complicated problem and a possible solution with 6 hyperplanes. Hyperplanes 4 and 5 are globally redundant whereas 2 and 3 are locally redundant.

There are different types of optimisations which can be performed. In fig. 23, hyperplanes 4 and 5 are redundant because they classify in the same way (up to a sign) all the patterns in the training set. This type of redundancy will be called global redundancy because the hyperplanes perform the same classification at the level of the entire training set. This type of redundancy can be eliminated by checking at the end of the training, whether there are two different hidden units which classify all patterns in the same way.

This type of redundant units are equivalent to the "noncontributing units" described in [Sietsma, 1991]. These noncontributing units are described as "units which [...] have outputs across the training set which mimic the outputs of another unit".

However, eliminating this type of redundancy as in [Sietsma, 1991] involves changing (by a precise amount) all the weights connected to the unit which will be preserved. In the constructive CBD approach, eliminating this type of redundancy at the end of the training would involve only removing the redundant unit and reconnecting all its outgoing weights to the other unit performing the same global classification. However, a much better redundancy elimination method will be presented shortly.

In the same fig. 23, hyperplanes 2 and 3 are only locally redundant. This means they perform the same separation in a limited region of the input space, in this case,

the positive half-space of hyperplane 1. However, the hyperplanes are not globally redundant because they classify patterns (a) and (b) differently. Consequently, a global search for redundant units cannot eliminate this type of redundancy.

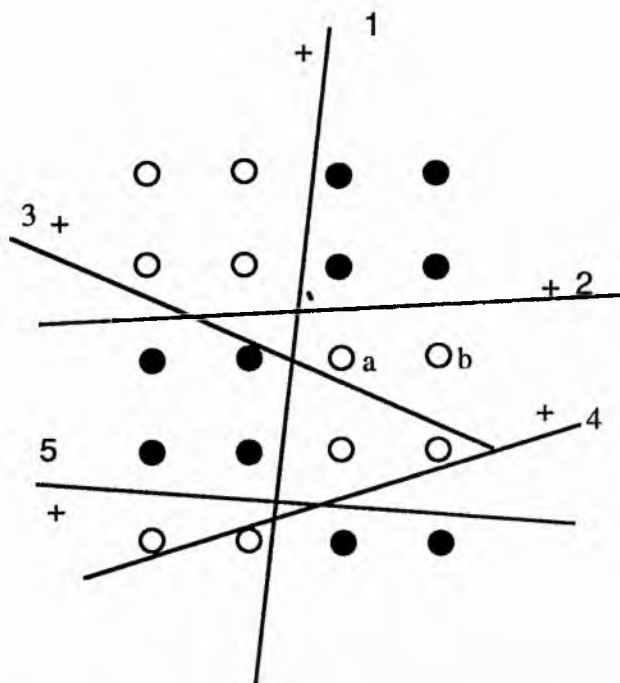


Fig. 24. The hyperplanes 4 and 5 are potentially redundant.

In fig. 24, hyperplanes 4 and 5 are potentially redundant in the sense that hyperplane 4 can be adjusted so that it performs the same global classification. Note that they are locally redundant in the area corresponding to  $(h_1, -)$  and  $(h_2, -)$ .

It is interesting to note that, in the case of the constructive CBD algorithm, other types of redundancy discussed in [Sietsma, 1991] such as units which have constant output across the training set are avoided by the algorithm itself. This is because the algorithm trains each unit so that it separates at least one unit and therefore the output will not be constant across the training set. The same can be said for "unnecessary-information units" which do not appear in the CBD constructive algorithm for the same reasons.

### 5.5.2 Eliminating redundancy

A straightforward approach is to optimise the position of a hyperplane with respect to all patterns and not only with respect to the patterns in the current region. Thus, after obtaining the best position of a hyperplane with respect to the region of the space in which the algorithm currently works, the hyperplane's position can be

optimised with respect to all patterns which haven't been correctly classified yet. Subsequently, each region determined by the new hyperplane (by splitting old regions crossed by it), is checked for consistency and labelled if possible.

This approach brings some difficulties. Firstly, the pattern set of a subgoal will be formed by two types of patterns. Suppose the algorithm is called in region R of the input space. The first type will contain the patterns contained in R. These patterns must be separated as well as possible because their separation ensures the convergence of the algorithm. The second type of patterns contains the patterns outside R whose separation is desired but not compulsory. Measures must be taken to ensure that the patterns outside R will not determine a depreciation of the classification score in R. For instance one could check the classification score in R after each weight change and stop when this score starts to decrease.

In order to improve the general efficiency, after a hyperplane is optimally positioned, the half-space with more patterns could be considered first for further separation of the patterns which are not separated yet. This would ensure that in the optimisation stage when all the patterns are taken into consideration, the possibility of the classification score in the current region R being worsened by the pattern outside R is diminished.

This feature offers a better use of the hidden units for the price of a slower training. The training speed is reduced for two reasons. Firstly, each subgoal training set will consider more patterns than in the standard CBD approach. Secondly, for each new hyperplane, all new regions determined by it must be checked for consistency and this could lead to a combinatorial explosion of the number of checks to be performed.

The redundancy elimination approach as discussed so far also tends to express the solution as a union of small regions even if a better expression (as only one bigger region) exists. This determines an inefficiency in the AND and OR layers. However, this problem can be solved by analysing (automatically) the expression generated and reducing it to its simplest form. After this reduction stage, the AND and OR layers can be designed.

For instance, the expression in fig. 25 can be put in a very simple form.

$$\begin{aligned}
& h_1 \bar{h}_2 h_3 h_4 + h_1 \bar{h}_2 \bar{h}_3 \bar{h}_4 + h_1 \bar{h}_2 \bar{h}_3 h_4 + h_1 \bar{h}_2 h_3 \bar{h}_4 = \\
& h_1 \bar{h}_2 h_3 (h_4 + \bar{h}_4) + h_1 \bar{h}_2 \bar{h}_3 \bar{h}_4 + h_1 \bar{h}_2 \bar{h}_3 h_4 = \\
& h_1 \bar{h}_2 h_3 + h_1 \bar{h}_2 \bar{h}_3 (h_4 + \bar{h}_4) = h_1 \bar{h}_2 h_3 + h_1 \bar{h}_2 \bar{h}_3 = \\
& h_1 \bar{h}_2 (h_3 + \bar{h}_3) = h_1 \bar{h}_2
\end{aligned}$$

Fig. 25. The reduction of a conjunctive form.

However, this redundancy elimination approach is not very good for all the reasons discussed above: more complicated treatment of the patterns, slower training and the solution needlessly expressed as a union of small regions. Furthermore, this approach waits until the training is finished before attempting to do anything. A much better approach is to check for redundancies during the construction of the network. Thus, precious training time can be spared and the solution can be obtained directly in a more compact form. Such an algorithm will be presented in the following.

Let us consider the problem presented in fig. 23. Let us suppose the first hyperplane introduced during the training is hyperplane 1. Its negative half-space is checked for consistency, found to be inconsistent and the algorithm will try to separate the patterns in this negative half-space (the others will be ignored for the moment). Then, hyperplane 2 will be put into position. Its positive half-space (intersected with the negative half-space of hyperplane 1) is consistent and will be labelled as a black region. The algorithm will now consider the region determined by the intersection of the negative half-spaces of 1 and 2, which is inconsistent. Hyperplane 4 will be added to separate the patterns in this region and the global solution for the negative half-space of 1 will be:

$$\bar{h}_1 h_2 + \bar{h}_1 \bar{h}_2 \bar{h}_4 \text{ is black and } \bar{h}_1 \bar{h}_2 h_4 \text{ is white.}$$

The situation after adding hyperplanes 1, 2 and 4 is presented in fig. 26. Then, the algorithm will consider the positive half-space of hyperplane 1 and will try to separate the pattern in this region. A new hyperplane will be introduced to separate the patterns in the positive half-space of hyperplane 1. Eventually, this hyperplane will end up between one of the groups of white patterns and the group of black patterns. Let us suppose it will end up between groups (a) and (b). This hyperplane will be redundant because hyperplane 2 could perform exactly the same task.



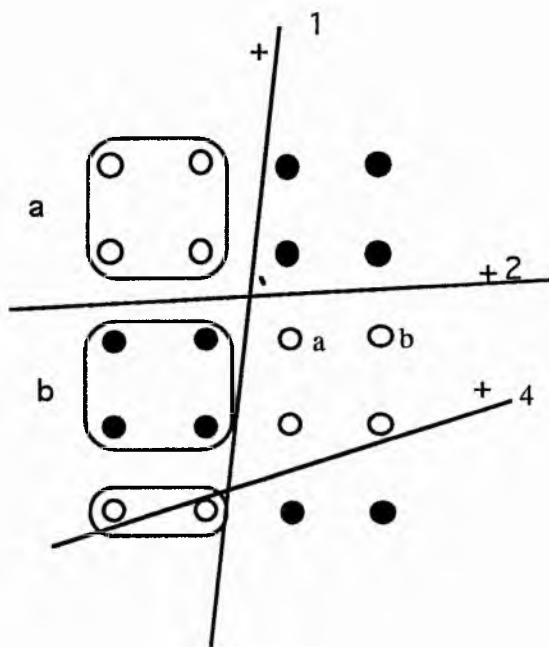


Fig. 26. Hyperplanes 2 and 4 separate the patterns in the negative half-space of hyperplane 1. A new hyperplane is needed for separating the patterns in the positive half-space of 1.

The redundancy is caused by fact that the algorithm takes into consideration only local pattern information i.e. only the patterns situated in the area currently under consideration, ignoring the others. At the same time, this is one of the essential features of the algorithm, the feature which ensures the convergence and yields a high training speed. Considering all the patterns in the training set is a source of problems as shown in the chapter on training (the herd effect for instance). One seems to face an insoluble question. Can an algorithm be local so that the training is easy and global so that too much redundancy is avoided?

The answer, at least for the constructive CBD algorithm is affirmative. The solution is to consider the patterns locally, as in standard CBD, but to take into consideration previous solutions as well. Thus, although the patterns are not considered globally which would make the problem difficult, some global information is used which will eliminate some redundancy from the final solution.

Let us reconsider separating the patterns in the positive half-space of hyperplane 1 with a new hyperplane between groups (a) and (b). Instead of automatically accepting this position, the algorithm could check whether there are other hyperplanes which classify in the same way the patterns in the positive half-space

of 1. In this case, hyperplane 2 does this and it will be used instead of a new hyperplane. Note that this does not affect in any way the previous partial solutions and therefore the convergence of the algorithm is still ensured. At the same time this modification ensures the elimination of both global and local redundancy and this is done without taking into consideration all patterns.

Computationally, this check needs only two passes (to cater for the possibly different signs) through the current subgoal. In each pass, the output of the candidate hyperplane unit is compared with the outputs of the existing units. If at the end of this, an existing unit is found to behave like the candidate unit, the existing unit will substitute the candidate unit which will be discarded.

This algorithm for redundancy elimination has been implemented and tested as an enhancement of the constructive CBD algorithm. The results will be presented in chapter 6.

In order to improve the efficiency of the algorithm further, one should consider the elimination of the potential redundancies i.e. those determined by two hyperplanes which are not redundant (either locally or globally) but could be so without affecting the solution. In other words, there is no perfect equivalent for the candidate hyperplane among the old hyperplanes but one of them could become such without affecting the previous solutions. For instance, in fig. 24, the old hyperplane 4 can substitute the candidate hyperplane 5 with a small change in its position and without affecting the separation of the patterns in the negative half-space of hyperplane 1 i.e. the region for which hyperplane 4 has been introduced.

In order to implement this, the redundancy elimination algorithm described could perhaps be further modified so that the old hyperplanes are not eliminated at the first pattern which is classified differently from the classification given by the candidate hyperplane, but only after further patterns have been classified differently. The number of further patterns taken might perhaps be determined by how important it is to eliminate the global redundancy for the given problem. If this could be done, at the end of the cycle through the patterns the set of old hyperplanes will contain hyperplanes which classify a majority of the patterns (in the current region) in the same way as the candidate hyperplane. Then, an attempt can be made to train these potentially redundant hyperplanes to classify correctly the patterns in the region they have been introduced for and to classify the patterns in the current region as the candidate hyperplane does. Once again, precautions must be taken to ensure that the optimisation training does not affect the classification of previously considered

patterns because this is the element which guarantees the convergence of the algorithm.

## **5.6. Generalisation properties. The influence of the order of the patterns**

### **5.6.1 General considerations**

The solution found by CBD must be considered from the point of view of the generalisation, as well. As shown in the chapter on generalisation, the assessment of the generalisation depends very much on the problem. The more one knows about the problem, the more additional information can be potentially fed into the network. However, well known algorithms like backpropagation do not give the user any possibility to feed this supplementary information into the network. The best the user can do is to train the network with the available patterns and compare the I/O mapping given by the network with the desired I/O mapping taking into consideration the contextual information available. In some cases, the user can add patterns to the training set in the hope that the I/O mapping which will be found by the network will get closer to the desired one. However, there is no clear possibility for the user to manipulate the patterns in order to influence the solution by bringing it towards the desired one.

The situation is the same for the known constructive algorithms. We were not able to find in the literature any well known possibility to control or at least influence the resulting I/O mapping by feeding into the network a priori information about the desired I/O mapping when such information is available.

In the following, it will be argued that such possibilities do exist for the CBD algorithm presented. It will be shown that the solution found by the CBD algorithm is influenced by the order of the patterns and this sensitivity can be used in a controlled manner to induce desired properties for the solution I/O mapping.

### **5.6.2 The influence of the order of the patterns**

The CBD algorithm solves the problem by dividing it into sub-problems according to a constraint based decomposition approach. Then, each sub-problem (defined by a subgoal) is solved. The network and the solution are constructed during the training by adding new units as required by subgoals. Therefore, both the final architecture of the network and the solution I/O mapping depend on the definition of the subgoals. In turn, the subgoals are constructed by taking patterns one by one

from the training set. It follows that the subgoals depend on the order of the patterns in the training set.

An important observation has to be made here. In general, in the neural network literature, the term "set of patterns" (or pattern set) is used to refer to the collection of patterns. This strongly suggests a connection with the mathematical term of set as a unordered collection of elements. However, both during the training and afterwards, the patterns are presented to the network one at a time i.e. serially. Due to this inherently serial training of a feed-forward network, a more appropriate term for the collection of patterns is that of "pattern sequence" or "training sequence". This term stresses the idea that not only the pattern themselves are important but their order as well.

Some algorithms use a static training sequence i.e. the patterns are presented to the network always in the same order. Some other algorithms permute randomly the patterns at the end of each training cycle such that the patterns are presented to the network in a sequence with a new order during each cycle. Sometimes, the same weight changing mechanism is used with or without permuting the patterns. It seems that most of the widely used training techniques do not have an intrinsic order of the patterns and this could justify perhaps the use of the term pattern set.

However, the constructive CBD algorithm presented is sensitive to the order of the patterns in the training sequence and this sensitivity can be used as an instrument to control the generalisation yielded by the solution, the resultant architecture and the training properties of the algorithm. The generalisation can be improved in the sense of bringing the solution given by the net to satisfy the assumptions (or knowledge) about the underlying function. The architecture can be improved in the sense of obtaining solutions with fewer hyperplanes and, in consequence, this could result in a shorter training.

Each type of problem has particular characteristics. Each training algorithm has some characteristics. If known, the characteristics of the problem, together with the characteristics of the algorithm can be used to influence the results.

Some characteristics of the constructive CBD algorithm presented, in solving a classification problem are: a) it constructs homogeneous regions with piece wise linear boundaries and b) for each hyperplane, it uses the patterns sequentially, in the order they are presented, to optimise the position of the hyperplane. Let us see how

one can use these characteristics to control to some extent, the solution and the training behaviour on a toy problem.

Let us consider the 2 grid problem presented in fig. 27.

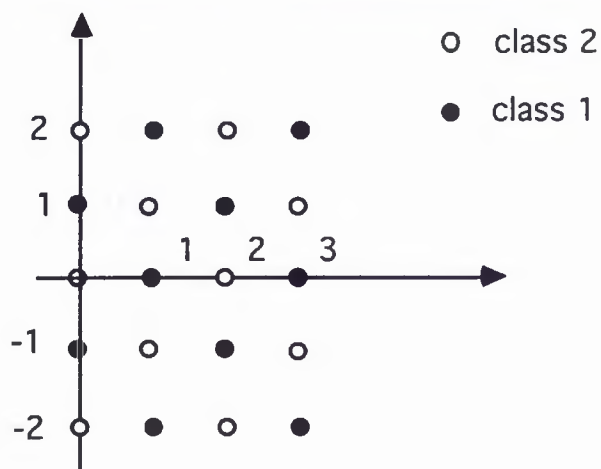


Fig. 27. The 2 grid problem with 20 patterns.

This problem, as a classification problem, admits various types of solution. Some possibilities are presented in fig. 28, 29 and 30. Note that all these solutions use the minimum number of hyperplane able to separate the classes and therefore, all of them are optimal solutions from this point of view. These solutions are very different. However, as it was shown in the chapter on generalisation, if no other information is given, none of these solutions can be seen as yielding bad or good generalisation.

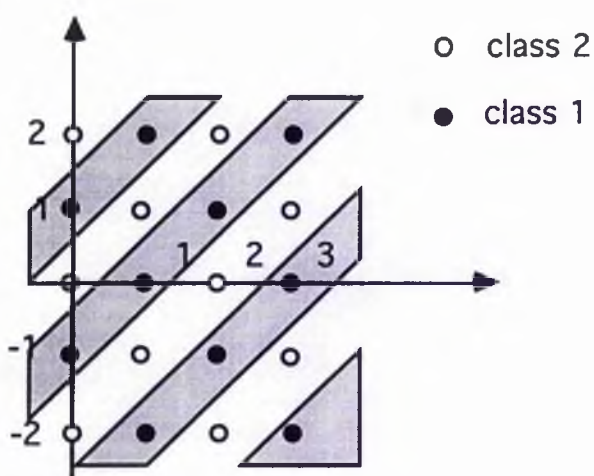


Fig. 28. A possible solution for the 2 grid problem.

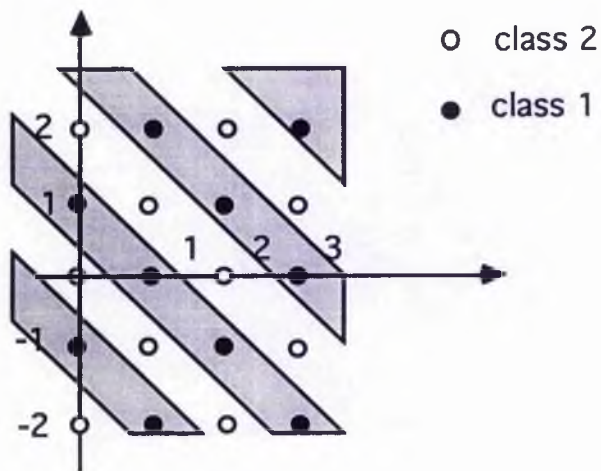


Fig. 29 A different possible solution for the 2 grid problem.

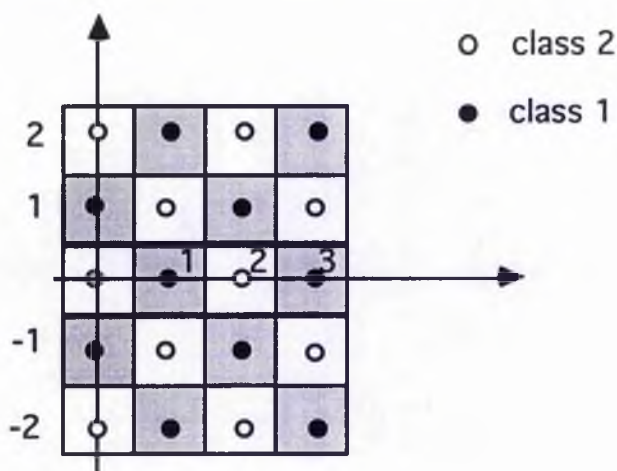


Fig. 30 Yet another possible solution for the 2 grid problem.

Now, let us assume there is some background information which says that a 'stripe' solution like those in fig. 28 or 29 is to be preferred to a 'checker-board' solution as in fig. 30 or to any other type of solution. Can one transmit this information to the network? Can one influence the solution?

The answer, at least for the constructive CBD algorithm presented is yes. Let us assume the preferred generalisation is the one in fig. 28. This solution is characterised by 7 straight lines with positive slopes. Each of them separates a 'stripe' of patterns from its neighbours. Let us recall the mechanism of the algorithm. The training takes place in subgoal sessions. For each subgoal, initially, one pattern from each class is chosen. This choice is not important for the convergence of the algorithm so the algorithm will pick up the first pattern of each

class in the order they appear in the training sequence. Subsequently, more pattern will be added one by one and the position of the current hyperplane will be modified so that new patterns are correctly classified as well, if possible.

We want the solution to have a hyperplane between the patterns (0,2) - white and (0,1) and (1,2) black. If we place these three patterns in this order in the pattern sequence, the solution will contain the desired hyperplane. This is because the first subgoal will contain one pattern of each class in the order they appear in the sequence i.e. (0,2) and (0,1) . A hyperplane will be used to separate them. The next subgoal will contain the set of three patterns (0,2) - white and (0,1) and (1,2) black. This subgoal will position the hyperplane in the desired position (see fig. 31). Subsequently, this hyperplane will not be moved anymore because all the other subgoals formed by adding another pattern to this set of three will be linearly inseparable and the patterns will be discarded without affecting the hyperplane.

The rest of the patterns (whose order was not important until now) can also be ordered so that the desired solution contains the other hyperplanes as desired. The position of the second hyperplane, for instance, can be determined (within the tolerance given by the patterns) if the next patterns are patterns #4, #5 and #6, as shown in fig. 31. The pattern sequence in which the patterns appear in the order presented in fig. 31 will determine the algorithm to construct a solution as the one in fig. 28. Such a sequence uses the order of the patterns to convey the message that a solution which puts patterns like (#2,#3), (#7,#8,#9,#10), etc. into the same homogeneous region is to be preferred.

Now, let us assume that a solution as the one in fig. 29 is desired. A pattern sequence which determines the algorithm to construct such a solution is presented in fig. 32.

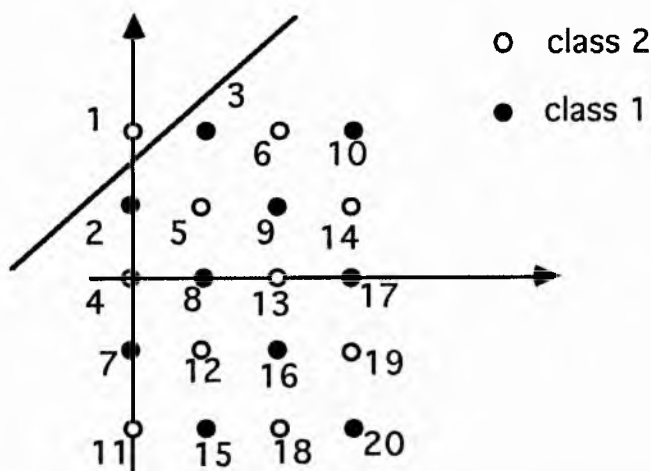


Fig. 31. Influencing the solution using the order of the patterns. If the patterns are ordered as shown by their labels, the first hyperplane will be forced into the position shown (within the tolerance given by the patterns) by the patterns #1, #2 and #3. The second hyperplane will be forced into position by the patterns #2, #3, #4 and #5.

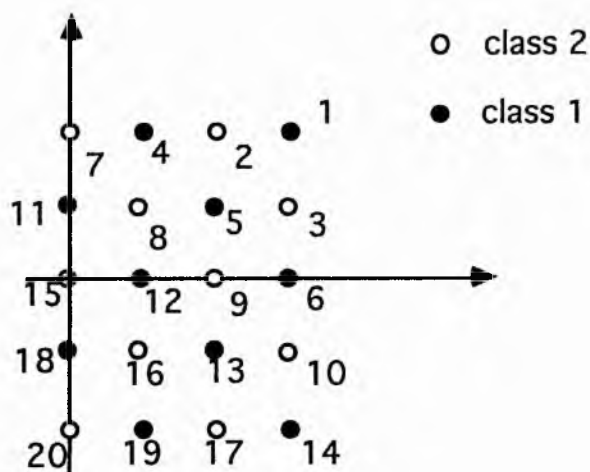


Fig. 32. Influencing the solution using the order of patterns. This ordering determines the algorithm to construct a solution like the one presented in fig. 29.

As an observation, it must be said that in this particular case, the ordering determines the algorithm to find a solution using the minimum number of hyperplanes necessary for this problem and this minimal solution can be obtained consistently, for any number of patterns in the training sequence.

The ordering which determine the desired solution for the 2-grid problem is not unique. Any ordering in which the patterns in the desired stripes (such as (#2, #3), (#4, #5, #6), (#7, #8, #9, #10), etc.) are permuted among themselves will determine



the same solution. The important condition for this problem is that all the pattern contained in one stripe come before any pattern in the following stripe. However, this is a sufficient condition. Perhaps, there are orderings which do not respect this condition and, for a particular choice of parameters will still yield the desired solution.

A question can be raised here as to the possibilities of practical use of this feature of the CBD algorithm. Can the user design the pattern sequence so that a certain type of solution is preferred by the network if the problem is less artificial and perhaps with an I/O mapping that is more difficult to understand? A general and simple way of designing the pattern sequence is to group together all patterns from the same class which are desired to be located in the same region. It is probable that this can be done even for complicated problems by clustering together inputs from the same class [Musavi, 1992].

In conclusion, the sensitivity of the constructive CBD algorithm described to the order in which the patterns are presented is a positive feature not a weakness. The user can use the ordering of the patterns in the training set to give the network information about a preferred type of solutions and to force the network to find such a solution. This possibility is specific to the constraint based decomposition among other training algorithms.

### **5.7. Other issues**

The main weakness of the constraint based decomposition algorithm is that the solution is not optimal in the sense that the solution is not always guaranteed to use the minimum number of hyperplanes able to separate the patterns. In a classification problem, for instance, CBD will give only a solution as opposed to the best solution. However, Golea and Marchand in [Golea, 1990] cite a result ([Hyafil, 1976]) showing that the derivation of the optimal decision tree is NP-complete. It can be shown that a network with the structure of the network constructed by the CBD algorithm is equivalent to a decision tree ([Sethi, 1990a]). Therefore, training and constructing a neural network with this structure and using the minimum number of hyperplanes is NP complete. If the NP completeness of the problem is taken into consideration, the fact that the solution is not optimal becomes less important as long as the solution is not extremely wasteful. A non-optimal solution, is better than no solution at all.

Another issue is the dependency of the constructive CBD algorithm presented on the initial weight state. The initial weight state can be very important for training algorithms as standard backpropagation. Particularly bad initial weight states can compromise the results of the training and particularly good initial weight states can speed up the training and improve generalisation [Denoeux, 1993]. Unfortunately, if the weight are initialised randomly, it seems that the probability of getting a particularly bad initial weight state is higher than the probability of getting a particularly good one. Therefore, this sensitivity of many training algorithms with respect to the initial weight state is rather a disadvantage than an advantage. Even if a particularly good initial weight state is used, the designing of such a weight state is not simple.

On the other hand, the constructive CBD algorithm is far less sensitive to the initial weight state than standard backpropagation or other constructive algorithms. This is because the initial weight state is always a weight state for a single hyperplane which will be trained with a linearly separable problem. A solution is guaranteed to exist and it is assumed that the chosen training algorithm for one layer is able to find such a solution. The position of the hyperplane after this first subgoal training is much more dependent on the patterns which are separated than on the initial position of the hyperplane. Subsequently, patterns are added and the hyperplane is moved until a final position is reached. From this, it is clear that the order of the patterns which influences directly the definition of the subgoals and which in turn modifies the position of a new hyperplane is far more important than the initial weight state.

At the most, the initial weight state influences the training time necessary for the first subgoal after a new hyperplane has been added.

The enhanced version of the CBD algorithm is presented in the following. This enhanced version contains the implementation of the redundancy and locking detection mechanisms.

**An enhanced CBD algorithm**

**separate** ( region, C1=set of patterns in C1, C2=set of patterns in C2, factor ) is

Build a subgoal S with patterns  $x_1^{C1}$  and  $x_1^{C2}$  taken at random from C1 and C2. Delete  $x_1^{C1}$  and  $x_1^{C2}$  from C1 and C2.

Add a hidden unit and train it to separate  $x_1^{C1}$  and  $x_1^{C2}$ . Let h be the hyperplane which separates them.

**For** each pattern p in C1 U C2.

    Add p to the current subgoal S

    Save h in h\_copy

    Train with the current subgoal S

**if not success then**

        Restore h from h\_copy

        Remove p from S

**For** each pattern p in C1 U C2                      /\* this is the check for global redundancy \*/

**For** each old\_hyperplane in old\_hp\_set

**if** p is classified differently by old\_h and h **then**

            /\* the hyperplanes h and old\_h are not redundant \*/

            remove old\_hyperplane from old\_hp\_set

**if** old\_hp\_set is not empty **then**

    /\* any of the hyperplanes in old\_hp\_set is redundant with h; pick up any of them \*/

    h = any of the elements of old\_hp\_set

**Let** new\_factor = factor **and** (h, '+')

**If** the positive half-space determined by new\_factor contains only patterns in the same class Cj **then**

        Classify new\_factor as Cj

**else**

        Delete from C1 and C2 all the patterns which are not in h+. Store the result in new\_C1 and new\_C2.

        Separate( h+, new\_factor, new\_C1, new\_C2, new\_factor )

**Let** new\_factor = factor **and** (h, '-')

**If** the negative half-space determined by new\_factor contains only patterns in the same class Cj **then**

        Classify new\_factor as Cj

**else**

        Delete from C1 and C2 all the patterns which are not in h-. Store the result in new\_C1 and new\_C2.

        Separate( h-, new\_factor, new\_C1, new\_C2, new\_factor )

## **CHAPTER 6**

### **More experiments with the constraint based decomposition constructive algorithm**

#### **6.1 Introduction**

This chapter presents some more experiments with the constraint based decomposition constructive algorithm. The main enhancements developed in chapter 5 are now tested and the enhanced algorithm is compared with the standard version.

Section §6.2 presents the hypotheses to be verified by experiments. Section §6.3 presents the method used to investigate these hypotheses and the sections §6.4, §6.5 and §6.6 present the experiments regarding locking detection, redundancy elimination and the influence of the order of the patterns respectively. Each such section has a sub-section summarising the conclusions of that set of experiments. The general conclusions regarding the CBD constructive algorithm are presented in section 6.7.

The experiments were performed on three problems. In the main body of the chapter, a number of experiments are presented in detail. These experiments are necessary and sufficient for the reader to reach the conclusions presented but they investigate the issues using just one problem. The results of the similar experiments for the other two problems are presented in appendix 1.

Appendix 2 contains a method which can be used to estimate a parameter used in some experiments, the locking tolerance.

#### **6.2 Hypothesis to be verified by experiments**

These experiments study the influence of various enhancements upon the training properties and the solution given by the constructive CBD algorithm. The enhancements investigated are locking detection, redundancy elimination and controlling the solution using the order of the patterns in the training set. These enhancements have been implemented and incorporated as options in the software program which simulates the training so that the effect of each of them can be studied independently. Further experiments with the CBD algorithm are described in [Draghici, 1995].

The characteristics followed were i) the number of hyperplanes used in the solution and ii) the training speed (in operations/connection crossings) for the locking detection and redundancy elimination enhancements. The influence of the order of the patterns has been investigated by studying the I/O mapping implemented by the solution found by the algorithm.

The order of the patterns is expected to influence the solution in a systematic way, allowing the user to select between different possible types of solution.

The locking detection is expected to bring an improvement of the training speed without affecting the number of hyperplanes used in the solution. The redundancy elimination is expected to reduce the number of hyperplanes used in the solution without affecting the training speed. These expectations, if confirmed, would allow the redundancy elimination and the locking detection to be used simultaneously, summing up their positive effects.

All enhancements have been experimented on 3 problems: the 2-spiral problem with 194 patterns (in  $6\pi = 3$  times around the origin), the 2-spiral problem with 770 patterns (for the same total length - 3 times around the origin) and the 2-grid problem. The 2-grid problem can be seen as an 2D extension of the XOR problem (the XOR patterns are included in the patterns of the 2-grid problem) and also as a 2D extension of the parity problem (in the 2-grid problem, the output should be 1 - black- if the sum of the Cartesian co-ordinates is odd). These three problems (the XOR problem, the parity problem and the 2-spiral problem) are considered difficult problems and are used as benchmarks due to their high degree of non-linearity. Thus, the performance of the technique on these problems can be seen yielding good information about the performance of the technique in general.

### **6.3 Methods used**

In order to investigate the hypothesis above, a number of experiments have been performed and their results have been processed statistically. A short explanation of the terms used and of the processing performed is given in the following. Sources like [Hugill, 1985] were used for choosing the statistical methods used and interpreting the results.

Most of the experiments performed involve two samples  $x_1, x_2, \dots, x_m$  and  $y_1, y_2, \dots, y_m$  of readings (for instance number of hyperplanes used in the solutions with and without redundancy reduction, number of operations used with

and without locking, etc.). It is assumed that these are random samples from two populations of possible readings. In these conditions, the "null hypothesis" states that the two populations are (in fact) the same. The "one-sided alternative hypothesis" is that the  $x$  values tend to be consistently bigger (or consistently smaller) than the  $y$  values. The statistical argument works by believing the null hypothesis until the value of a certain test statistic is so extreme that would be very unlikely under the null hypothesis. In these conditions, one is forced to change one's mind and accept the alternative hypothesis.

This argument however, does not guarantee the falsity of the null hypothesis. The only claim of this argument is that the null hypothesis is unlikely to be true. The probability of rejecting the null hypothesis when it is, in fact, true is called the "significance level" or "confidence level" and is usually taken to be 5%. The confidence level of 5% has been used in all tests performed.

In order to compare the effects of a particular enhancement, a number of tests were performed using the algorithm with and without the enhancement. Each individual experiment was run in the same conditions for the two versions: the same initial weight state, order of the patterns, weight changing mechanism, etc. In these conditions, an appropriate statistical test is the paired two-sample  $t$ -test for means (small samples). This test uses the  $t$ -distribution (Student's  $t$ -distribution) and tests whether the means of the two samples are significantly different. The test assumes the populations from which the samples have been drawn have normal distributions<sup>1</sup> with the same variance. The test uses the samples to calculate the value of the  $t$  variable. The value resulted from this calculation is compared with the critical value of the  $t$  for the given confidence level. The comparison decides whether the null hypothesis can be rejected or not.

However, the influence of the order of the patterns could not be investigated in this way because the constant number of hyperplanes employed by the solution when a good ordering of the patterns was used makes the normality assumption unacceptable. In these conditions, the results of the experiments were analysed by

---

<sup>1</sup>The normality assumption is necessary because the number of the readings in the two samples was relatively small (20). However, checks performed on the data showed that this normality assumption is reasonable.

plotting the frequency with which a given number of hyperplanes was used in various trials, with and without a good ordering.

## **6.4 Locking detection**

### **6.4.1 Experiments with locking detection**

Let us reconsider the definition of a locking situation. A locking situation is a situation in which the existing patterns determine the position of the hyperplane within a given tolerance. In such a situation, the hyperplane cannot be moved anymore (outside the tolerance) without misclassifying some existing patterns. From this definition it is clear that such a situation may or may not appear frequently in a problem. Consequently, the locking detection may or may not bring an important improvement to the training of a particular problem. Therefore, one expects this mechanism to be more important for some problems and less important for others. In order to assess correctly the efficiency of this mechanism, a problem of each category has been chosen.

A problem for which the locking detection mechanism was expected to bring an important improvement is the 2 spiral problem (proposed originally by Wieland and cited in [Lang, 1988]). This is because this problem is characterised by a relatively large number of patterns in the training set, a high degree of non-linearity and therefore needs a solution with a relatively large number of hyperplanes. Due to the distribution of the patterns, it was expected that some hyperplanes be locked into position by some close patterns. The results of 20 trials using 20 different orderings of the patterns in the training set containing 194 patterns are presented in fig. 33. The average number of operations (connection crossings) used by the variation without locking was 65,340,103.5 whereas the average number of operations (connection crossings) used by the variation with locking detection was 32,192,110.8. This corresponds to an average improvement of 50.73% of the training speed.

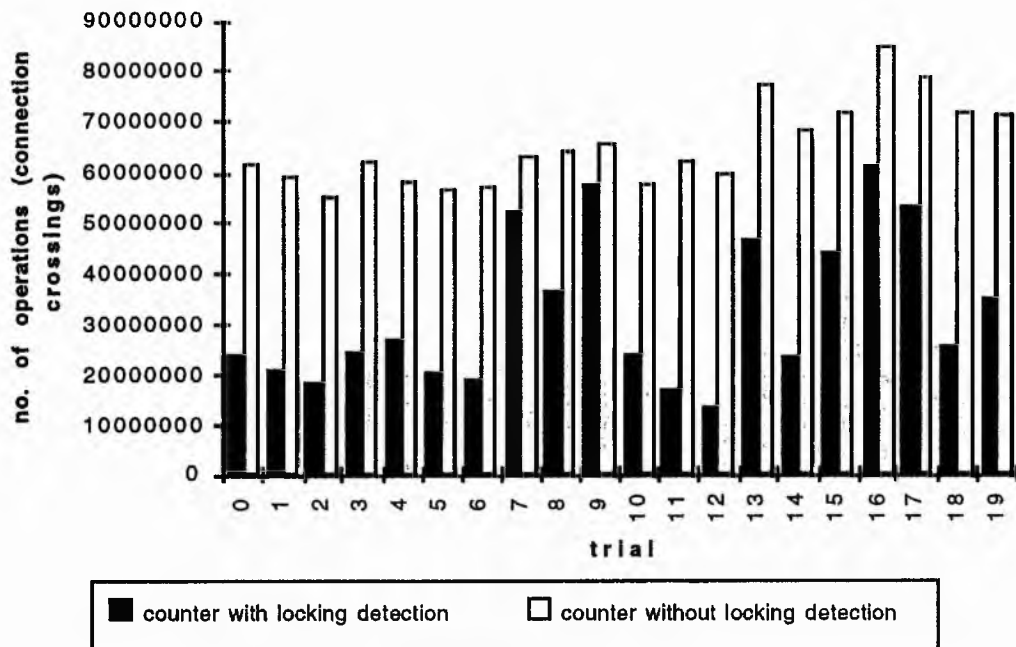


Fig. 33 Comparison between the number of operations (connection crossings) used by the technique with and without the locking detection in solving the 2-spiral problem (194 patterns).

The data used for the paired 2-sample t-test and the results of the test are presented in table 1. The test was performed on the number of operations used in 20 training sessions with and without the locking detection mechanism. The calculated value of  $t$  is -14.97 and is smaller than the critical value of  $t$  for 5% level of confidence which is -1.72. In these conditions, the null hypothesis can be rejected. In other words, the improvement brought by the locking detection mechanism in terms of convergence speed is significant. The same test was performed on data regarding the number of hyperplanes. The data used and the results of the test are presented in table 2. The calculated value of  $t$  is -0.84 and is smaller than the critical value of  $t$  (5%) which is 2.09 for the two-sided alternative hypothesis. In these conditions one can conclude that the locking detection mechanism does not affect significantly the number of hyperplanes used in the solution.

The results of 16 trials using 16 different orderings of the patterns in the dense training set of the 2-spiral problem (containing 770 patterns) are presented in fig. 34. The average number of operations (connection crossings) used by the variation without locking was 869,269,999.7 whereas the average number of operations (connection crossings) used by the variation with locking detection was



117,714,120.4. This corresponds to an average improvement of 79.56% of the training speed. The data used and the results of the t-test for this problem are presented in appendix 1. The t-test shows the improvement is considerable and cannot be due to statistical phenomena (the probability of rejecting the null hypothesis when it is true is of order  $10^{-9}$ ).

counter with locking	counter without locking	t-Test: Paired Two-Sample for Means		
23873451	61969152		locking	no locking
20941140	59092200	Mean	32192110.8	65340103.5
18576321	55240515	Variance	2.24E+14	6.70E+13
24451530	62225490	Observations	20	20
27235689	58375758	Pearson Correlation	0.7352534	
20091675	56656830	Pooled Variance	9.01E+13	
19037715	56973318	Hypothesized Mean Difference	0	
52037046	63108396	df	19	
36572916	64489581	t	<b>-14.07059</b>	
57594450	65521344	P(T<=t) one-tail	8.43E-12	
24158889	57717225	t Critical one-tail	<b>1.7291313</b>	
17050188	62267904	P(T<=t) two-tail	1.69E-11	
13164408	59831412	t Critical two-tail	2.0930247	
46783668	77502405			
23511867	68264094			
43958337	71526108			
61018854	84594792			
53480301	78484008			
25548990	71574414			
34754781	71387124			

Table 1. The data (no. of operations) used for the paired 2-sample t-test for the 2-spiral problem with 194 patterns and the results of the test. Because  $t = -14.97 < t_{\text{critical}} (5\%) = -1.72$ , the test proves the fact that the improvement brought by the locking detection mechanism in terms of the number of operations is significant.

no of hp's with locking	no. of hp without locking	t-Test: Paired Two-Sample for Means		
34	34		locking	no locking
63	61	Mean	58.8	59.7
60	63	Variance	48.9052632	59.3789474
61	59	Observations	20	20
62	68	Pearson Correlation	0.79286893	
68	67	Pooled Variance	42.7263158	
61	60	Hypothesized Mean Difference	0	
59	57	df	19	
56	62	t	<b>-0.8423441</b>	
51	59	P(T<=t) one-tail	0.20503686	
58	60	t Critical one-tail	1.72913133	
61	61	P(T<=t) two-tail	0.41007372	
58	66	t Critical two-tail	<b>2.0930247</b>	
59	53			
67	59			
62	63			
56	50			
56	58			
63	64			
61	70			

Table 2. The data (no. of hyperplanes) used for the paired 2-sample t-test for the 2-spiral problem with 194 patterns and the results of the test. Because  $t = -0.84 > t_{\text{critical}} (5\%) = -2.09$ , the test proves the fact that the locking detection mechanism does not affect significantly the number of hyperplanes used in the solution<sup>2</sup>.

---

<sup>2</sup>Strictly speaking, the test does not prove the fact that the locking detection mechanism does not affect significantly the number of hyperplanes used in the solution. The test merely shows that there is no evidence to sustain that the locking does affect the number of hyperplanes. This word of caution is valid for the rest of the experiments involving negative statements, as well.

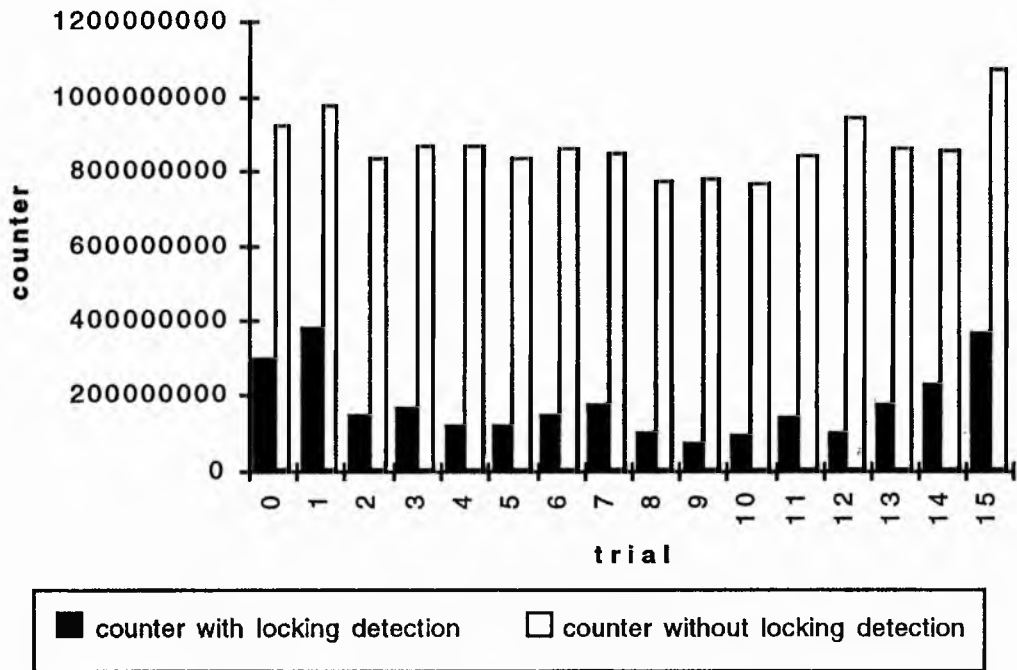


Fig. 34 Comparison between the number of operations (connection crossings) used by the technique with and without the locking detection in solving the 2-spiral problem (770 patterns).

A problem for which the locking detection mechanism was not expected to bring such a spectacular improvement is the 2-grid problem presented in fig. 35. This problem is characterised by a relatively small number of patterns in a training set which are relatively sparsely distributed in the input space. If one looks at a solution, one can note that in general, the patterns do not determine the position of the hyperplanes within a tolerance comparable to the tolerance which characterises the 2-spiral problem. In other words, it is probable that locking situations (with the same tolerance) appear less frequently during the training. In these conditions, it is normal for the version with the locking detection to show less improvement over the standard version of the algorithm. The results of 20 trials using 20 different orderings of the patterns in the training set are presented in fig. 36. The average number of operations (connection crossings) used by the variation without locking was 1,149,094.2 whereas the average number of operations (connection crossings) used by the variation with locking detection was 1,038,254.1. This corresponds to an average improvement of 9.65%. Although important even for this problem, the improvement brought by the locking detection mechanism is not as spectacular as for the 2-spiral case. The data used and the results of the t-test for this problem are

presented in appendix 1. The t-test shows that the improvement is statistically significant for the given confidence level even for this problem.

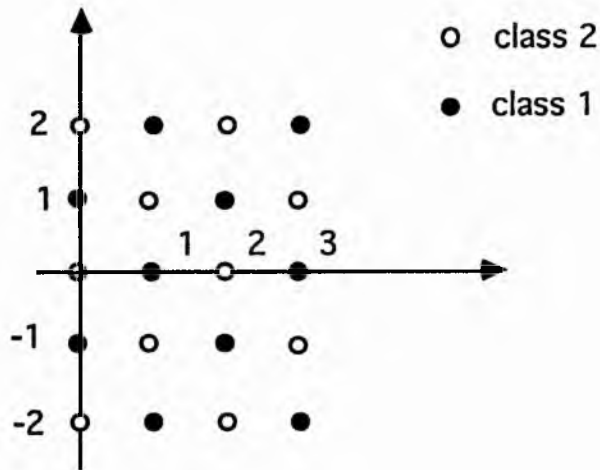


Fig. 35. The 2 grid problem with 20 patterns. The XOR patterns are included in these 20 patterns.

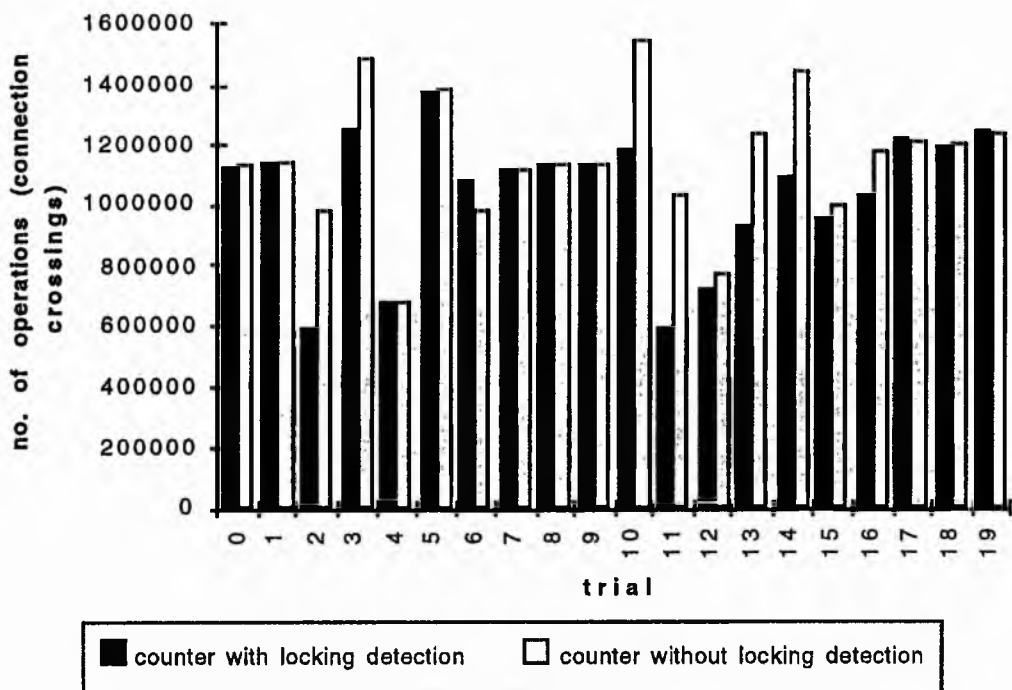


Fig. 36 Comparison between the number of operations (connection crossings) used by the technique with and without the locking detection in solving the 2-grid problem (20 patterns)

### **6.4.2 Conclusions of the locking detection experiments**

The following conclusions can be drawn from the experiments presented above.

1. In standard CBD training, a significant amount of time is spent wastefully trying to modify the position of a hyperplane whose position is determined within a small tolerance by some of the input patterns.
2. The ratio between this wasted time and the time spent usefully depends on the problem and increases with the density of patterns and the degree of non-linearity of the problem. This is because the higher the density of patterns, the more patterns in a given area and therefore, the greater the probability of a locking situation to appear.
3. The locking detection mechanism can improve the training speed by eliminating those subgoal training sessions which cannot improve any further the position of a hyperplane. All experiments performed showed that: i) the effect of the locking detection mechanism on the training speed is significant (even dramatic for some problems) and ii) the effect of the locking detection mechanism on the number of hyperplanes used is not significant.

## **6.5 Redundancy elimination**

### **6.5.1 Experiments with redundancy elimination**

As the constraint based decomposition technique is more sensitive to the order of the patterns in the training set than to the initial weight state, 20 trials with different orderings of the patterns in the pattern set were performed with and without checking for redundant hyperplanes. The results are presented in fig. 37. The pattern set is that of the 2-spiral problem and contains 194 patterns. The order of the patterns in the training set was changed randomly before each trial. For each trial, the same random permutation of the patterns in the pattern set was used for both the standard and the enhanced version of the algorithm. The standard version of the algorithm solved the problem with an average number of 87.65 hyperplanes (the average is performed over the 20 trials). The enhanced version of the algorithm with the redundancy check solved the same problem with an average of 58.8 hyperplanes which represents an average improvement of 32.92%.

The t-test performed on the number of hyperplane data coming from experiments with this problem shows that the effect of the redundancy elimination mechanism

upon the number of patterns used in the solution is significant ( $t = -18.95 < t_{\text{critical}} = -1.72$ ). The data used and the results of the test are presented in table 3. The same test performed on the number of operations data shows that the redundancy elimination mechanism does not affect significantly the results from this point of view. These data and the results of the test are presented in table 4.

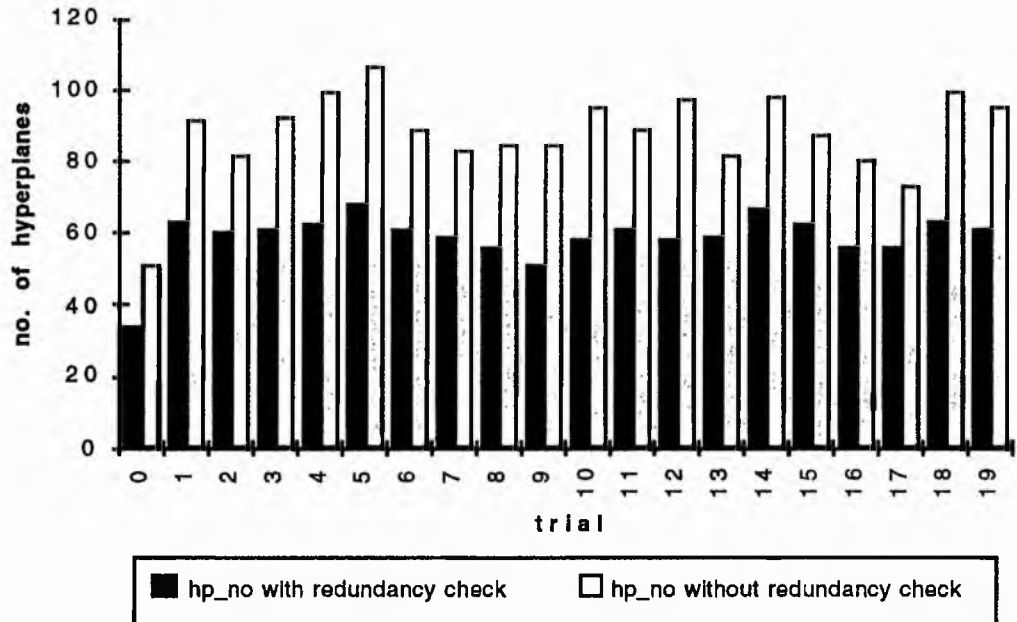


Fig. 37. Comparison between the number of hyperplanes (hidden units on the first layer) used by the technique with and without the check for redundancy. The training set is that of the 2-spiral problem containing 194 patterns.

hp with red	hp no red	t-Test: Paired Two-Sample for Means		
34	51		hp with red	hp no red
63	91	Mean	58.8	87.65
60	81	Variance	48.9052632	141.186842
61	92	Observations	20	20
62	99	Pearson Correlation	0.86495795	
68	106	Pooled Variance	71.8736842	
61	89	Hypothesized Mean Difference	0	
59	83	df	19	
56	84	t	<b>-18.95224</b>	
51	84	P(T<=t) one-tail	4.23E-14	
58	95	t Critical one-tail	<b>1.7291313</b>	
61	88	P(T<=t) two-tail	8.45E-14	
58	97	t Critical two-tail	2.0930247	
59	81			
67	98			
62	87			
56	80			
56	73			
63	99			
61	95			

Table 3. The data (no. of hyperplanes) used for the paired 2-sample t-test for the 2-spiral problem with 194 patterns and the results of the test. The test shows that the redundancy elimination has a significant effect on the number of hyperplanes used.



counter red	counter no red	t-Test: Paired Two-Sample for Means		
23873451	23859378		counter red	counter no red
20941140	21110946	Mean	32192110.8	32997516.8
18576321	32630958	Variance	2.24E+14	1.95E+14
24451530	21448674	Observations	20	20
27235689	17293392	Pearson Correlation	0.91323957	
20091675	25493598	Pooled Variance	1.91E+14	
19037715	20166777	Hypothesized Mean Difference	0	
52037046	52022517	df	19	
36572916	47468874	t	<b>-0.589672</b>	
57594450	48205260	P(T<=t) one-tail	0.28117893	
24158889	18493518	t Critical one-tail	1.72913133	
17050188	19486491	P(T<=t) two-tail	0.56235787	
13164408	16845498	t Critical two-tail	<b>2.0930247</b>	
46783668	46210368			
23511867	32392617			
43958337	45252810			
61018854	59444958			
53480301	52506951			
25548990	29842692			
34754781	29774058			

Table 4. The data (no. of operations) used for the paired 2-sample t-test for the 2-spiral problem with 194 patterns and the results of the test. The test shows that the redundancy elimination mechanism does not affect significantly the number of operations.

The algorithm was also tested on a pattern set of the same 2-spiral problem containing 770 patterns. The results are summarised in fig. 38. The standard version of the algorithm solved the problem with an average number of 186.50 hyperplanes (the average is performed over 16 trials). The enhanced version of the algorithm with the redundancy check solved the same problem with an average of 99.19 hyperplanes which represents an average improvement of 46.82%. The data

used and the results of the t-test for this problem are presented in appendix 1. The t-test shows that the improvement is statistically significant for the given confidence level.

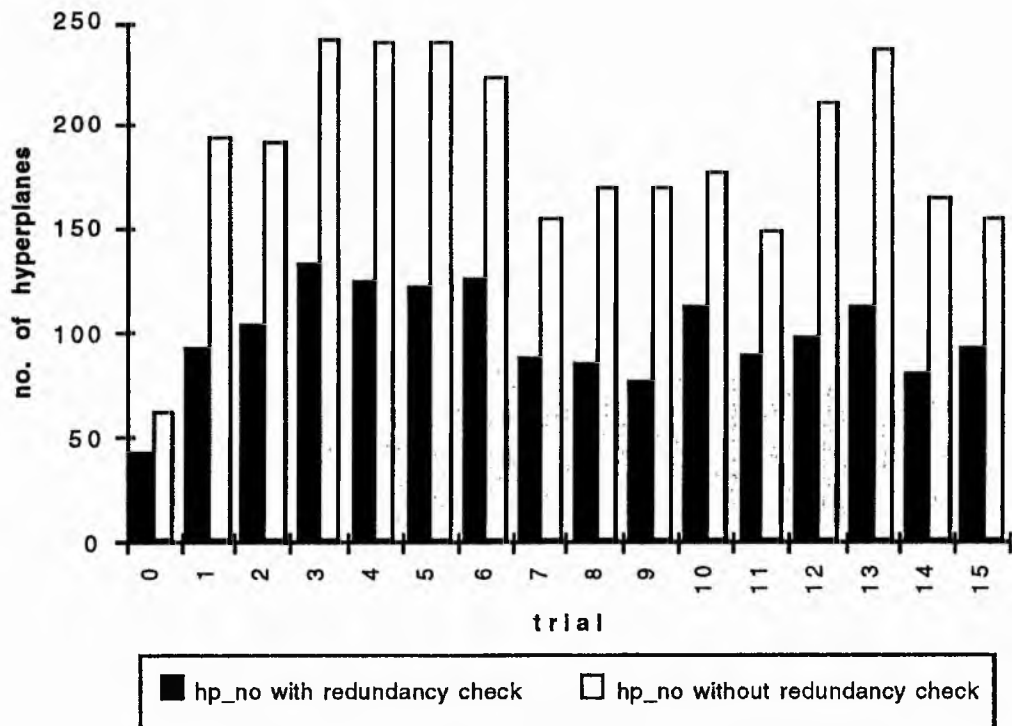


Fig. 38. Comparison between the number of hyperplanes (hidden units on the first layer) used by the technique with and without the check for redundancy. The training set is that of the 2-spiral problem containing 770 patterns.

The same comparison was performed for the 2 grid problem. The results of the experiments (the numbers of hyperplanes used in the solution) are presented in fig. 39. The standard version of the algorithm solved the problem with an average number of 12.45 hyperplanes (the average is performed over the 20 trials). The enhanced version of the algorithm with the redundancy check solved the same problem with an average of 11.05 hyperplanes which represents an average improvement of 11.24%. The data used and the results of the t-test for this problem are presented in appendix 1. The t-test shows that the improvement is statistically significant for the given confidence level.

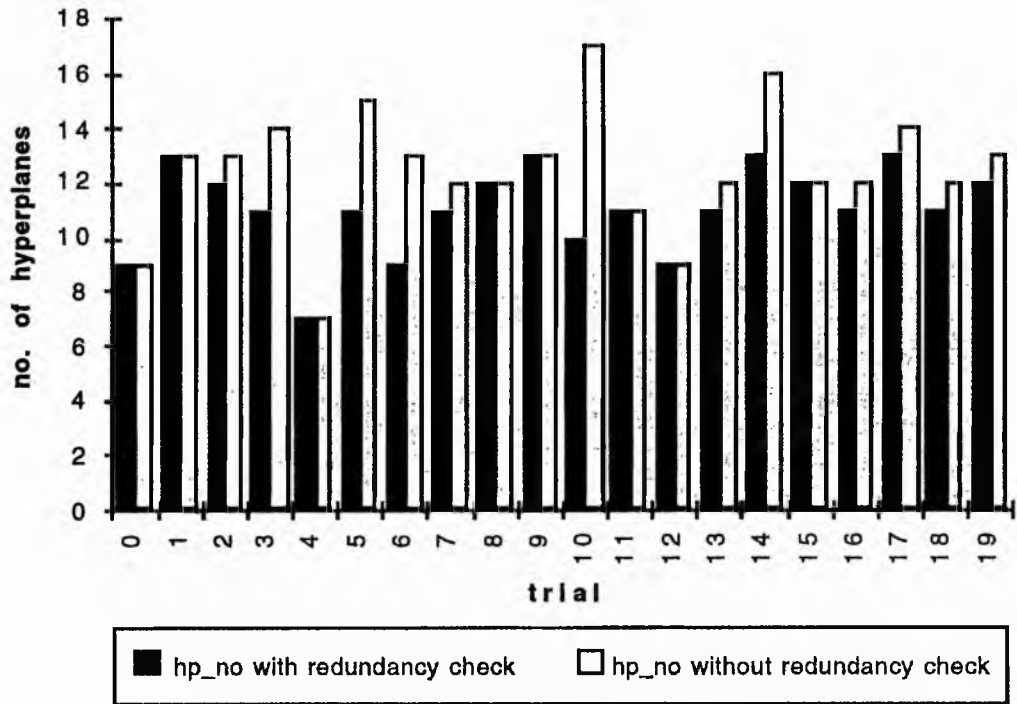


Fig. 39. Comparison between the number of hyperplanes (hidden units on the first layer) used by the technique with and without the check for redundancy. The training set is that of the 2-grids problem containing 20 patterns.

### 6.5.2 Conclusions of the redundancy elimination experiments

The following conclusions can be drawn from the experiments presented above.

1. The standard implementation of CBD, as for other constructivist techniques, tends to use an excessive number of units of which many are redundant.
2. The proposed mechanism for redundancy elimination was shown to be effective in different types of problems.
3. All experiments performed showed that: i) the effect of the redundancy elimination mechanism on the number of hyperplanes used is significant (even dramatic for some problems) and ii) the effect of the redundancy elimination mechanism on the number of operations needed is negligible.

## **6.6 The influence of the order of patterns**

### **6.6.1 Study of the influence of the order of the patterns**

The 2-grid problem with 20 patterns was chosen for investigating the influence of the order of the patterns because:

- i) The number and position of patterns are such that very different I/O mappings are possible.
- ii) The number of patterns is not too large and therefore the functioning of the algorithm can be easily followed and understood.

The purpose of these experiments is to show that in the case of the CBD algorithm presented, the order of the patterns in the training sequence can be used to convey supplementary information about the desired shape of the solution. Thus, the shape of the resulting I/O mapping can be controlled by the user.

Two sets of experiments were performed. The trials in the first set (20 trials) used a random order of the patterns in the training set. The trials in the second set (5 trials) used "good" orderings i.e. orderings which were designed (as explained in 5.6.1 and 5.6.2) to yield a solution "like" the one in chapter 5, fig. 28. A solution S was considered to be like the one in fig. 28 if all groups of patterns which are in the same region in fig. 28 were in the same region in S (i.e. the separating hyperplanes do not have to be parallel like in fig. 28 but they must form "stripes").

Both sets of trials were performed with the standard algorithm enhanced with the locking detection and the redundancy elimination mechanisms. The number of hyperplanes used in the solution are presented in table 5. The identical number of hyperplanes used by the technique in all tests performed on a pattern set arranged in a good order makes the normality assumption unacceptable. Therefore, the t test cannot be used anymore. In order to show the fact that the influence of the order of the patterns is significant, the frequency of a number of hyperplanes in the solution was plotted for the trials with a random and with a good order of the patterns. The results are presented in fig. 40. All trials performed on pattern sets in which the patterns were arranged in a good order yielded solutions with the minimum number of hyperplanes (7 in this case). The trials with a random order of the patterns used a variable number of hyperplanes.

All trials performed with the good order yielded the desired form of I/O mapping and found a solution using the minimum number of hyperplanes. An example of such solution is presented in fig. 41. The fact that all trials produced solutions as the desired one shows that the order of the patterns was effective in forcing the network to choose the desired generalisation.

random order	good order
red & lock	red & lock
9	7
1 3	7
1 2	7
1 1	7
7	7
1 1	
9	
1 1	
1 2	
1 3	
1 0	
1 1	
9	
1 1	
1 3	
1 2	
1 1	
1 3	
1 1	
1 2	

Table 5. Comparison between the number of hyperplanes used in the solution when patterns were presented in a random order and in a good order.

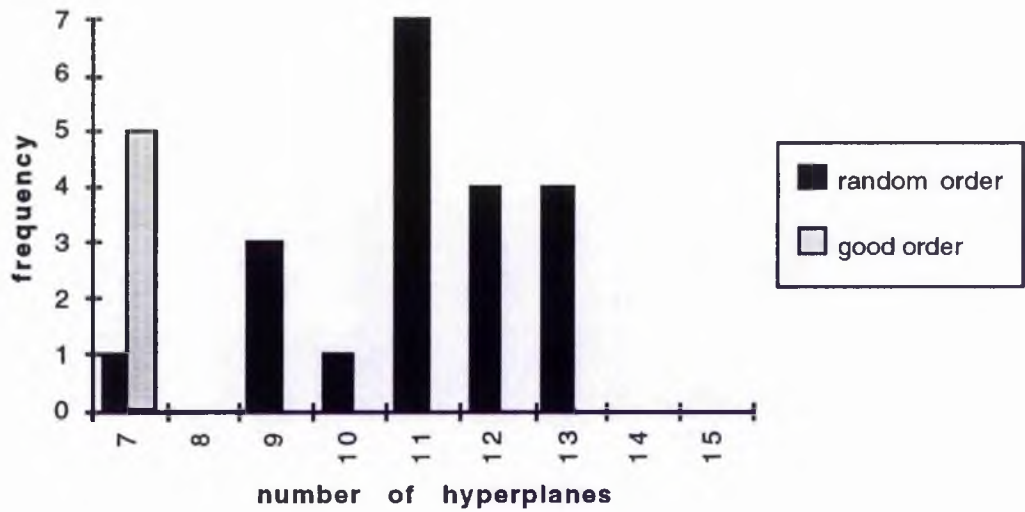


Fig. 40 The frequency of appearance of a given number of hyperplanes in the solution. The good order yielded 5 solutions with 7 hyperplanes whereas the random order yielded solutions with various number of hyperplanes.

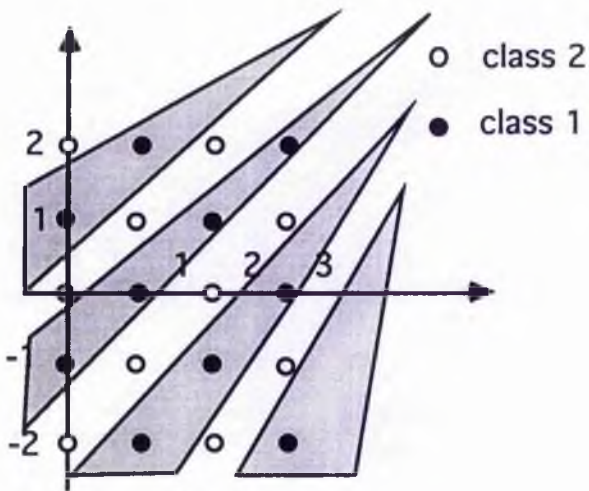


Fig. 41. An example of a solution having the desired characteristics.

#### 6.6.2. Conclusions for the experiments investigating the influence of the order of the patterns.

The following conclusions can be drawn from the experiments presented above.

1. The order of the patterns in the training sequence affects the solution given by the CBD algorithm presented.
2. This sensitivity can be used constructively to convey supplementary information about the desired type of solution and thus to determine the algorithm to favour it.

### **6.7. General conclusions**

Some drawbacks of the CBD algorithm presented have been discussed and some enhancements designed to eliminate these drawbacks have been presented. These enhancements are:

1. The locking detection mechanism - designed to improve the training speed by eliminating those subgoal training sessions which can not modify the current position of the current hyperplane.
2. The redundancy elimination mechanism - designed to eliminate hyperplanes which perform the same local classification.
3. The non-separability elimination mechanism - designed to eliminate those subgoal training which try to solve a linearly inseparable problem with a single hyperplane.

Of the above enhancements, the locking detection mechanism and the redundancy elimination mechanism have been implemented and shown to be effective. The linear separability mechanism has not been implemented.

The experiments performed did not show any negative effect of either the locking detection mechanism or the redundancy mechanism upon the resulting I/O mapping.

The CBD algorithm was shown to be sensitive to the order of the patterns in the training sequence. This sensitivity was shown to be an advantage because it can be used to control the shape of the solution. Experiments with a simple problem showed that this control is effective.

# CHAPTER 7

## Complex Backpropagation

### 7.1. Introduction

Section §7.2 presents the motivation behind the complex backpropagation network and algorithm. Section §7.3 presents the approach to the problem, analyses the requirements of this approach and identifies the necessary operators. The complex network is presented in section §7.4. A general overview of a device able to satisfy the requirements is given after which, a detailed analysis of the various options is given. In the end of this section, the processing performed by the network with the choices discussed is presented.

Section §7.5 presents the complex backpropagation training algorithm for the network presented. Other work on the complex backpropagation algorithm is presented in section §7.6 and the relation between this work and the network and algorithm presented in this thesis is discussed. The conclusions are presented in section §7.7.

The detailed derivation of the equations of the complex backpropagation in the framework presented is given in appendix 3.

### 7.2. Motivation: automatic selection of the type of model we want.

For the purpose of this discussion it is convenient to see a neural network as an approximating mechanism. In this framework, an underlying function (or phenomenon) is sampled, the network is trained with the samples and the task of the network is to approximate as well as possible the underlying function or phenomenon.

In the chapter on generalisation it has been shown that no matter how many samples there are, there are always many functions which pass through the given points and that without any other information, there are no reasons to consider any of them as being better than any other. A possible approach taking this into account is to have a very dense training set so that the desired function is sufficiently determined for the purpose of the specific application. In other words, for a very dense training set the difference between any function satisfying the training set (i.e. passing through the training points) and the underlying function is less than the error limit. This



approach can be unsatisfactory because the training of a large set of patterns is usually difficult.

A more elegant approach would be to feed the network not only with the data points but also with some information about the shape of the function to be approximated. If the shape of this underlying function is not known, the net could be fed with information about the shape of the desired function i.e. the shape of the function we would like the net to use in building the approximation.

Let us consider the example presented in fig. 1. Two different types of functions can be fitted through the same data points. As mentioned above, one approach is to feed the network with enough data points until the shape of a function passing through the points cannot be very different from the shape of the desired function.

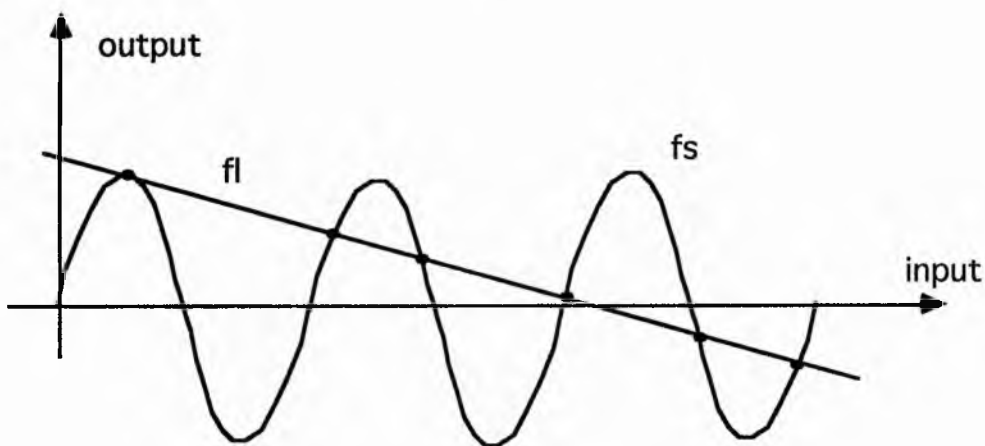


Fig. 1 Two different types of functions which pass through the same data points.

Instead of increasing the number of samples, one could feed into the net some information regarding the shape of the desired function. For instance, one could specify that the desired function should be a sinusoid. The net will take into consideration this and will look for a sinusoidal function instead of a linear one which could have been the straightforward choice. In this approach, the training of a neural network can be seen as a constraint satisfaction problem: find a function which passes through the given points whilst satisfying the constraints about the shape of the function.

The information regarding the shape can be fed into the network if the training problem is changed a little. Instead of associating values, the network could be asked to associate signals. By varying the shape of the input signal, the user should be able to control the shape of the output function. Thus, although the samples can

be the same, the generalisation properties of the network (i.e. the output values for the input values between the training points) can be controlled by feeding different signals as the input signal.

The motivation given by this signal approach requests the network to be able to associate input signals to output signals. As a signal is an infinite continuous variation of values, an appropriate finite representation must be found for the signals.

### **7.2.1. Parametric representation of signals**

Although there are neural network models and implementations which work with signals, this is rather the exception. Most neural networks paradigms deal with a finite number of I/O patterns. As stated in the previous section, in general, a signal is an infinite continuous variation of values. Clearly, a signal in this form cannot be used easily in a neural network framework. Even if the continuous sequence of values is sampled to obtain a discrete sequence, there is little one can do with an infinite set of values. A more suitable representation for a signal is needed.

A possible solution is to use a parametric representation of a signal. Such a parametric representation could contain all the information in the infinite continuous set of values which is the signal in the same way the analytical expression of a function can describe completely the function. This is natural because a signal is a function.

Thus, instead of working with the signals as infinite sets of values, the network can use their parametric representation: the parametric representation of the input signal can be associated to the parametric representation of the output signal. In this manner, the network will still associate a finite number of values but the interpretation of these values will be different. They will not be instantaneous values but parameters describing signals.

Let us consider for instance, the signal in fig. 2.

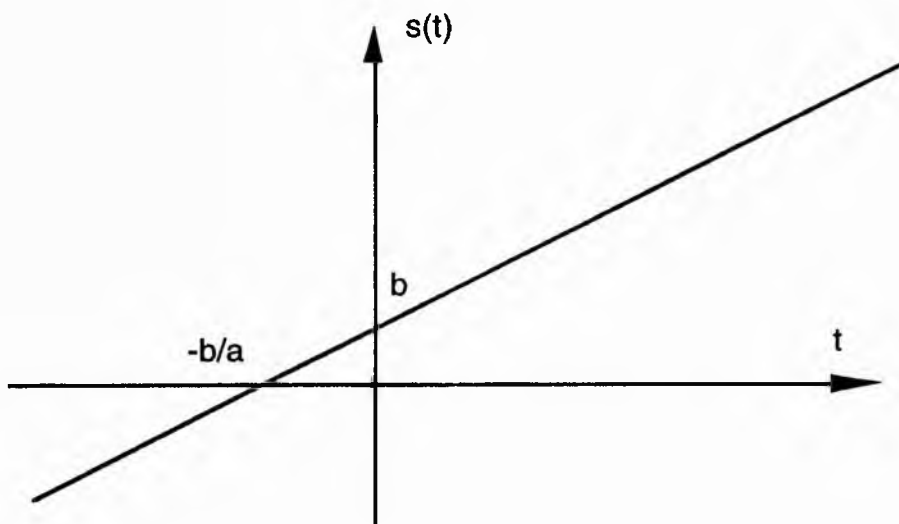


Fig. 2. A linear signal  $s(t) = a*t + b$ .

Instead of trying to store the infinite set of  $(t, s(t))$  pairs which form the signal one could store only the values  $a$  and  $b$  plus some information regarding the way the parameters  $a$  and  $b$  are used to construct the signal. Let us assume the desired output signal is a different linear function  $y(t) = c*t + d$ . In this case, the network could implement the signal association by associating the values  $(a, b)$  to the values  $(c, d)$ . The values  $(c, d)$  plus the contextual information that the signal is polynomial suffice to obtain any instantaneous value of the desired output signal for arbitrary  $t$ .

### 7.2.2. Signals determined by the training cycle

One can consider a training session using a fixed pattern presentation algorithm. In other words, the patterns are presented to the network always in the same order. In this situation, one could consider the signals given by the variation of the values over the training cycle. For instance, the variation of the input values over the training cycle determines a periodic input signal, the variation of the output values over the training cycle determines a periodic output signal and so on. The net could be trained to associate the input signals to the output signals.

Let us consider for instance the classical example of the XOR problem. The problem is defined in a I/O space with 2 inputs and 1 output. The input/output value associations are presented in table 1.

input 1	input 2	output
0	0	0
0	1	1
1	0	1
1	1	0

Table 1. The input/output value associations for the XOR problem

If the patterns are always presented to the network in the order given in the table, the following signals can be defined over the training cycle:

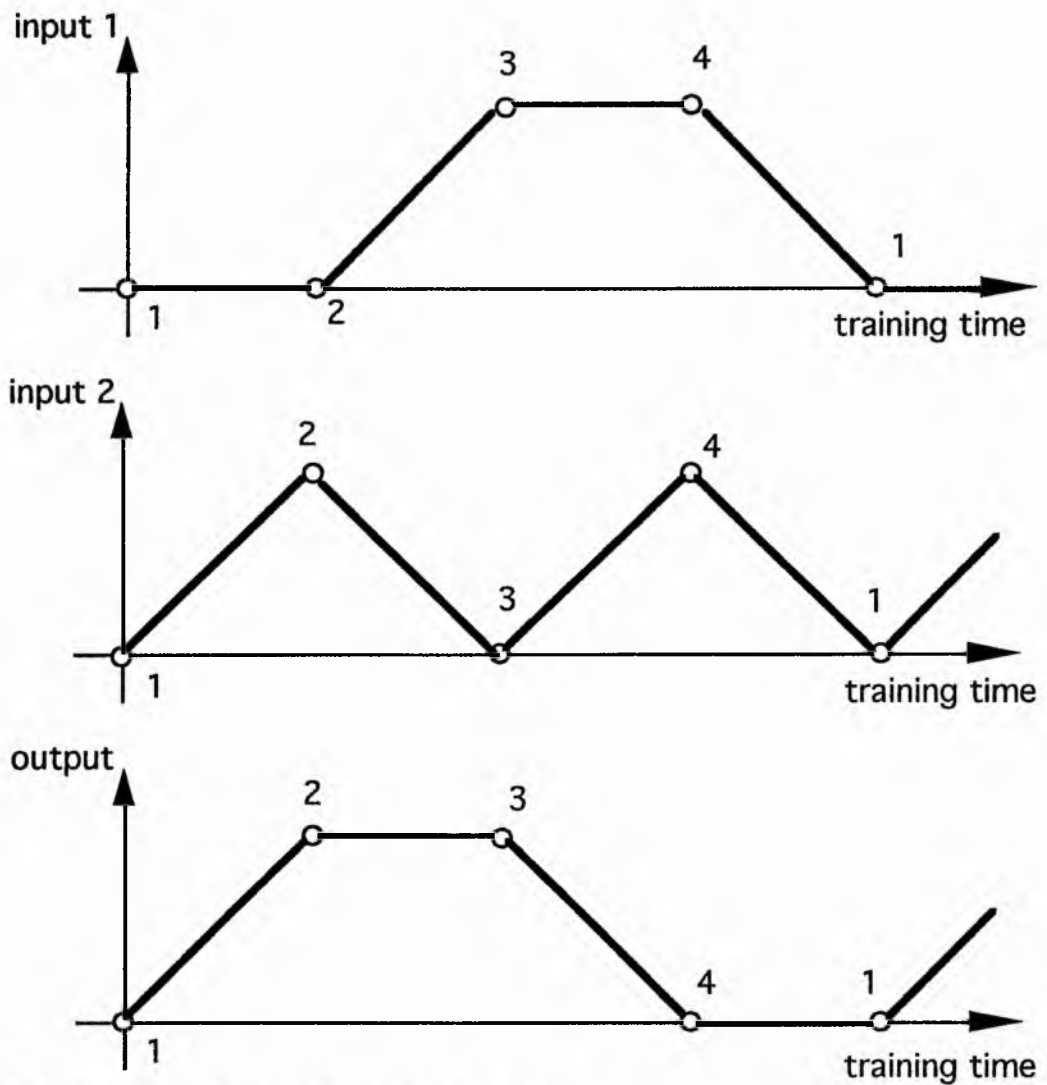


Fig. 2. Signals as defined by the training cycle.

A network which produces the output signal when fed with the two input signals in fig. 2 would implement XOR because, for each valid (i.e. 0 or 1) input combination, the output will have the correct value.

A disadvantage of this type of approach is that the network is able to respond correctly to the input patterns (assuming that the training was successful) only if they are presented in the order they had in the training set. For this reason, one could say that such a network can do less than a standard network trained with individual values. This is because the standard network is able to respond correctly to the input patterns presented in any order. If both training sessions are successful, the quantity of information stored in a network trained with individual values (as in the standard backpropagation approach) is greater than the quantity of information stored in the network trained with signals determined by the training cycle.

One can analyse the problem from a different point of view though. According to this different approach, a network trained with a signal will have stored more information than the network trained with values because it has also learned the whole *path* in-between the training points. Although this alternate point of view can be valid for some applications, the first approach will be preferred in this thesis. This is because it is assumed that the important aspect is to learn the I/O surface (from some of its samples) and not to store a particular path through it.

For a network trained with signals defined by the training cycle to be equivalent (in terms of random access to the instantaneous values) to a network trained with instantaneous values, it is necessary to train the network with all signals determined by permuting the patterns in the training set. For anything else than toy problems, this become highly unfeasible.

This seems to be an insurmountable obstacle. However, the question is whether arbitrary value association abilities are always necessary. Usually, what is more important than arbitrary value associations is the shape of the I/O surface. If the shape of the I/O surface is the desired one, one can eventually determine the correct result for a given input independent of the moment (and thus the order) when this input was presented to the network. A possible method for doing this is to define some standard paths in the input space. Thus, the input space can be scanned in some standard way for instance in the way an electron beam scans the screen of a TV set. Then, the patterns can be presented to the network in the order given by this standard scan. Later on, when a new pattern comes, the same scan can be used to

determine its neighbours and its position in the ordering and thus define the input signal.

Chen in [Chen, 1992] uses such an approach to implement a mapping from input space to a time dimension. In this approach, the network has a weight state for each moment in time. When a new input value comes, the input space is scanned in an a priori defined way to determine the time value corresponding to the new input. In [Chen, 1992], a time value is associated with each concentric circle centred in the origin. Thus, for a new input, the distance from the input to the origin is used to select a time value which in turn, is used to select the appropriate weight state. If a weight state does not exist for the calculated time value, two closest existing time values are selected. The weight states which correspond to these closest time values are used to calculate an interpolated weight state for the intermediate time value associated with the given input.

This standard scan approach shows that as long as the information regarding the desired I/O function is stored somehow in the trained network, it can be used to obtain valid outputs even for untrained inputs, although perhaps with some extra computation.

### **7.2.3. Arbitrary signal association**

A different approach is to consider the possibility of arbitrary signal associations. In this approach the input/output entities to be associated to each other are signals and they come directly from the underlying phenomenon. Since the inputs/outputs are already signals, the user does not need to fit their own signals through some input and output samples as in the previous case. In this approach, the network performs a mapping from a signal space to another (the same or different) signal space.

This approach can be more or less useful than the previous one, depending on the problem. A system trained with values will not be able to perform one-many associations of instantaneous values whereas a system trained with signals could do it. This is because the same instantaneous value can be preceded by different values thus distinguishing it from other occurrences of the same value. A system associating signals would be more efficient if the input patterns appear always in the same order for instance. This is because in this case the random access at instantaneous values which is the flexibility introduced by the value association approach is not needed. On the other hand, if all one needs is a value association between input and output without any one-many problems, and if the input values

do not appear always in a preferential order, a network trained with values is a more sensible choice.

### 7.3. Approach

#### 7.3.1. Signal space. A signal as a vector in signal space.

Let us consider a periodic function  $f$  with the period  $T_0$ . The Fourier series of the function  $f$  is given by the expression:

$$f(t) = \frac{a_0}{2} + \sum_{n=1}^{\infty} [a_n \cos(2\pi n f_0 t) + \sin(2\pi n f_0 t)] \quad (1)$$

where  $f_0$  is the fundamental frequency given by the inverse of the period  $1/T_0$ . The function  $f(t)$  which is defined by a continuous infinite set of instantaneous values can be completely (without loss of information) represented by the set of parameters  $\{f_0, a_0, a_1, \dots, a_k\}$ . Let us consider a space in which each dimension is associated with a term in the sum above. We shall call this signal space. In this signal space, any function of time can be represented as a point. This point has the co-ordinate along each axis equal to the Fourier coefficient of the term associated with that axis. The signal space can be seen as a space with  $2n$  dimensions: for each harmonic  $n$  we have two axes, one for sin and one for cos.

Any point in the signal space can be represented in an equivalent fashion as a vector drawn from the origin to the given point. The sum of two signals can be seen now as the vectorial sum of the vectors representing each of the given signals.

The Fourier representation of a function (1) is just a possibility within the parametric representation approach whose scope is much larger. Any parametric representation could be used with appropriate network structures. The Fourier representation will be used in the following to illustrate the way a network can be tailored to the needs on a particular parametric representation.

#### 7.3.2. Principal Component Analysis

The general idea of this approach is to reduce the dimensionality of the data space so that the processing becomes easier. Sometimes, there exists a set of directions so that the projection of the original data onto this set of directions can be done without too much loss of information. The interesting situation is when this set of directions contains fewer dimensions than the original. In statistics this technique is known as

principal component analysis (PCA). There are neural networks architectures and training algorithms which perform PCA (Oja, Sanders, see [Hertz, 1991]).

The PCA idea can be applied in at least two ways. Firstly, one could try to eliminate some dimensions which are not useful for the characterisation of the data. For instance, if data points from two classes have the same co-ordinate along a certain direction of the input space, that particular dimension can be eliminated without any prejudice to the possibilities of discriminating between classes. Certainly, this is a particular case in which the data is contained in a subspace of the original input space. Also, in general, the data points are not contained in a subspace of the original space. Furthermore, in general the principal component directions are not directions of the reference system. Nevertheless, sometimes is possible to project the data into a space with fewer dimensions without losing information.

The second way the PCA idea can be applied is to perform a dimensionality reduction with the minimum loss of information. In such a situation, the number  $N$  of dimensions of the representation space is given and the system should choose  $N$  directions so that the loss of information is minimised.

A similar idea can be used in the signal space defined above. For simplicity, let us consider that the dimensionality reduction is minimal i.e. only one dimension has to be eliminated. Given a vector in a  $n$  dimensional space, choose its representation in an  $n-1$  dimensional space so that the error introduced by this dimensionality reduction is minimal. In fig. 2, the 3-D vector  $v$  can be approximated by one of its projections onto the 2-D subspaces  $xy$  or  $xz$ . The projection onto  $xy$  is  $v_1$  and the projection onto  $xz$  is  $v_2$ . In this situation, the error  $e_1$  determined by approximating  $v$  with  $v_1$  is much larger than the error  $e_2$  determined by approximating  $v$  with  $v_2$ . In these conditions, it is clear that the  $y$  axis can be eliminated with smaller loss of information than the  $z$  axis.



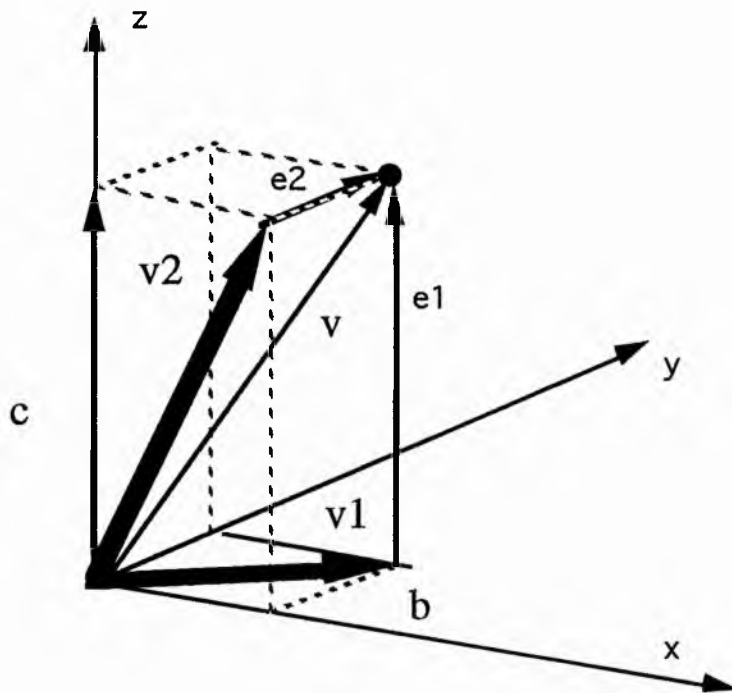


Fig. 2 The error introduced by eliminating a dimension depends very much on which dimension is eliminated.  $V$  can be approximated by both  $V1$  and  $V2$ .

### 7.3.3. Projections in signal space and their Fourier interpretation. Operators in signal space: rotate and scale.

A projection in signal space is equivalent to ignoring some terms from the Fourier series of the signal. The error can be minimised by considering only the most important terms i.e. the terms with the largest coefficients.

In order to perform an association of two signals in signal space, a network should be able to build the output signal from the input signal by using the information stored in its weights. This is exactly what a standard network (associating values) does. In the case of a value association, the input values are presented to the network and propagated forwards through various layers. At the level of each layer, the values are multiplied by the weights and passed through the activation functions of the neurons until, eventually, the desired output values are obtained at the level of the output neurons.

Something similar should be done even in the case in which the network operates with signals. The network should take the representation of the input signal in the signal space propagate it through the neurons and build the signal space

representation of the output signal by using the information stored in the network's weights.

As shown, the input signal and the output signal can be seen as vectors in signal space with the origin in the origin of the co-ordinate system. In order to transform the input vector into any output vector (for the simple case of associating a single output signal to a single input signal) the network should be able to rotate and scale a vector. Using these two operations, any input vector can be associated with any output vector if the magnitudes of the rotation and scaling operations are chosen properly (see fig. 3).

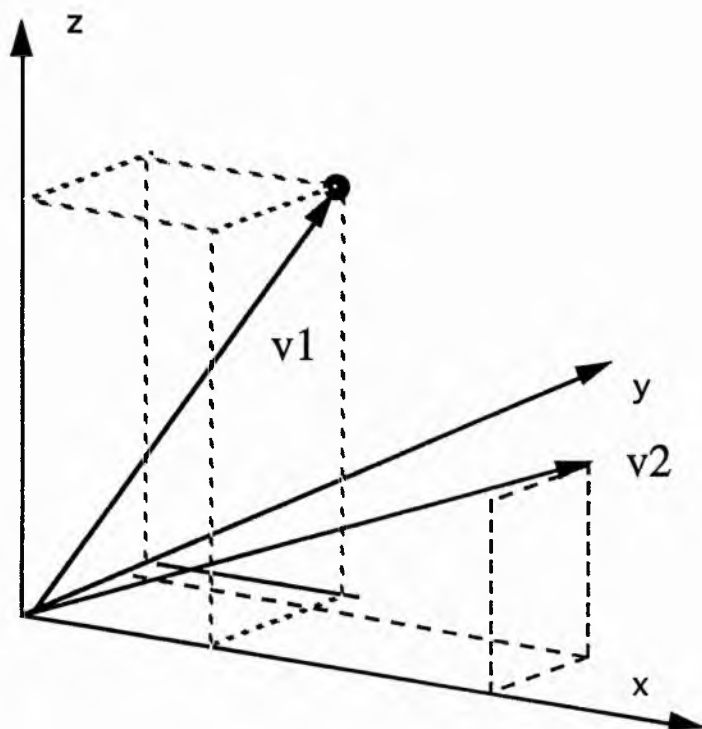


Fig. 3. The vector  $v_2$  can be obtained from  $v_1$  by performing a rotation around the origin and a scaling given that the magnitudes of the rotation and scaling are properly chosen.

#### 7.4. The complex network

The main purpose of our device is to associate input signals to output signals. A sketch of such a device is presented in fig. 4. The input signal is pre-processed and a parametric representation of it is calculated. This parametric representation of the input signal is then fed into the processing unit which could be a neural net. The processing units uses the parametric representation of the input signal and the

information stored in the processing unit itself (during the training) to construct the parametric representation of the output signal. In general, different representations can be needed for the input and the output so a different number of parameters can be used for the input and the output signals. Eventually, the parametric representation of the output signal is post-processed to build up the output signal in the desired form.

The parametric representation of a signal can be the vectorial representation in signal space. Thus, any infinite signal can be represented using a finite number of parameters, the co-ordinates of the vector in signal space. For those signals which have an infinite number of terms in their Fourier representation, an arbitrarily good approximation can be constructed by taking into consideration a finite but sufficient number of terms i.e. a sufficient number of components of the vector in signal space.

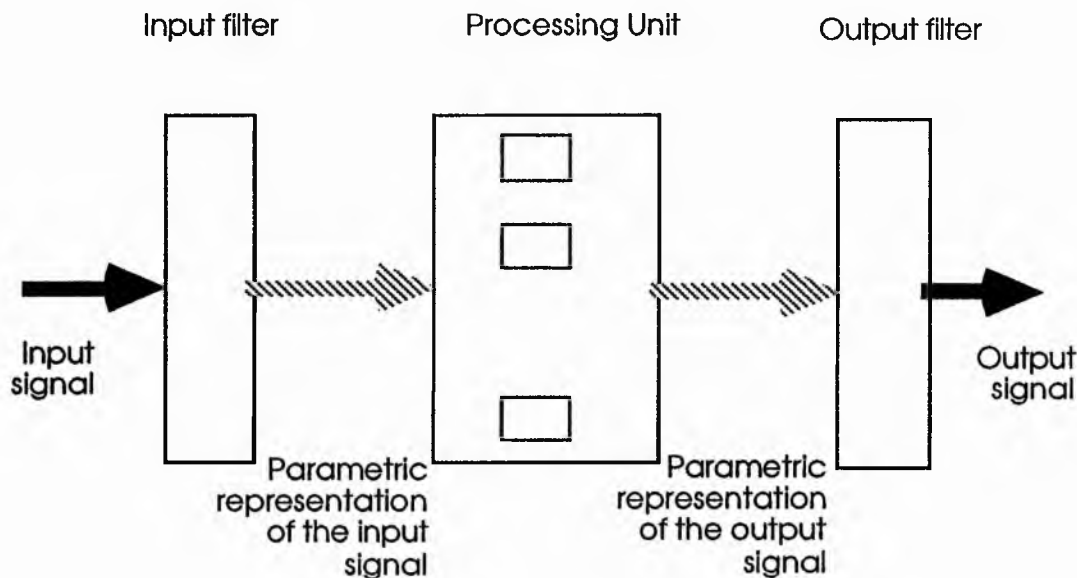


Fig. 4. A conceptual schema of a device associating signals.

We shall consider each element in the above schema.

#### 7.4.1. The pre-processing filter.

The main purpose of the pre-processing filter is to put the input in a form manageable by the processing unit.

There are two possibilities depending on the form of the raw input and the purpose of the whole net.

1. The input is in a form of instantaneous values. The net must preserve the ability to retrieve the correct output for individual input values. As shown, a standard scan of the input space must be chosen, and from it, a proper representation of the input values must be built.

2. The input is a signal. The net must associate the input signal to a given output signal. In this case, the main problem is to build a finite parametric representation of the input signal which can be fed to the processing unit.

#### **7.4.2. The post-processing filter.**

The role of the post-processing filter is to revert from the internal representation of the output signal to the form of the output signal which is requested by the application. This transformation should be the inverse of the transformation implemented by the pre-processing filter. Thus, if the processing unit is to perform just the identical transformation or alternatively if the input filter is to be connected directly to the output filter, the input signal should be found unchanged at the output of the output filter.

#### **7.4.3. The processing unit.**

The processing unit must be able to transform the parametric representation of the input signal in the parametric representation of the output signal by using information stored in the learning stage.

If a neural network is to be used as a processing unit, the type of the network, the representation of the signal and the processing performed by the units in the network should be carefully designed in accordance to the problem, the approach used and the overall purpose of the network. These issues will be discussed in the next two sections.

##### **7.4.3.1. The approach.**

In this approach, all the available samples of the input signal are used to construct a parametric representation able to be fed to the processing unit. The input signal can be a raw input signal coming from the underlying phenomenon or a signal defined by scanning the input space in predefined mode. The approach is called the parallel approach because the information in the samples is processed in a global way, as if all the samples were processed in parallel. A requirement for this approach is that all

the samples of the signal or in other words, all the instantaneous values be available at the same time so that a finite parametric representation can be calculated.

Let us suppose the input signal is represented as a vector in signal space. This vector has non-zero co-ordinates along  $n$  directions from the set of directions of the signal space. The output signal can be seen as another vector in signal space. This output vector has non-zero co-ordinates along  $m$  directions of the signal space. In the general case, these  $m$  directions are different. The processing unit must be able to associate the parametric representation of the input signal i.e. the  $n$  non-zero co-ordinates of the vector in signal space corresponding to the input signal to the parametric representation of the output signal i.e. to the  $m$  non-zero co-ordinates of the vector corresponding to the output signal.

A possibility for implementing this processing unit is a neural network using complex weights. In this case, the projection of a signal vector along a direction from the co-ordinate system of the signal space can be represented by a complex quantity in radius/phase form. The radius channel can convey information about the magnitude of the components (projections of the original signal along directions of the co-ordinate system of the signal space) and the phase channel can convey information about the axes themselves. In this framework, the interpretation of a pair (radius, phase) is: the original signal has a component along the axis corresponding to "phase" and the magnitude of that component is "radius". A phase weight should modify the axis of a component i.e. perform a rotation in signal space whereas the radius weight should modify the length of the component. A weight (radius and phase) can redirect a component of the input signal along a different direction and change its length at the same time in such a way that it becomes a component of the output signal.

#### **7.4.3.2. The designing of the processing unit.**

There is a crucial question to be asked here: how do we use the neurons so that the processing of the net makes sense?

In a neural net, each neuron acts upon its input on three levels: weights, excitation and transfer function. A few different possibilities will be discussed at each level.

**Level 1: weights.** The choice to be made here regards the way in which the weights modify the incoming excitation.

There are at least three options. The first option is to alter separately the radius and the phase values by multiplying them correspondingly with radius and phase weights. The phase weight should be integer to allow for a physical interpretation of the signal space approach. On the other hand, if the weights are integers, one cannot apply a gradient technique which needs continuous variations and derivatives.

A second option is to allow real phase weights while still modifying separately the radius and the phase values. This allows for the application of gradient techniques but loses the immediate physical interpretation of the quantities in the network. Since the phases are not integers after they are affected by the phase weights, the signals can not be seen as components along an orthogonal co-ordinate system.

However, the multiplication of phase values with the phase weights can still be seen as a rotation along the origin. The difference is that, if the weights are not integer values, the signal is not directed along one of the axes of the rectangular co-ordinate system but along a different direction which is not orthogonal on the initial direction given by the phase previous to the multiplication with the weights.

Another different option is to multiply the incoming excitation and the weights as two complex numbers. From a computational point of view, this option is equivalent with the first one in the sense that for any set of radial and phase weights used according to the first option, there exists another set so that the complex quantity resulted after the modification by the weights according to the independent multiplication rule and to the complex multiplication rule are the same.

The only difference between the first and the third option is that the first option modifies the phase by multiplying it with the phase weight whereas the third option modifies it by adding the phase weight to it. From the point of view of the possibilities, both options have the same properties in the sense that both offer access to all complex values. In these conditions, the first option is preferred because it is expected to offer a similarity of the behaviour of the radial and phase components during the gradient training.

**Example.** Let us suppose the output of the unit on the previous layer is  $(r, \varphi)$  where  $r$  is the radius and  $\varphi$  is the phase. Let us suppose the weight is  $(w_r, w_\varphi)$  where  $w_r$  is the radial component of the weight and  $w_\varphi$  is the phase component of the weight.

The first option is to limit  $w_\phi$  to integer values and to calculate the incoming excitation (modified by the weight) as  $(r^*w_r, \phi^*w_\phi)$ .

The second option is to calculate the incoming excitation in the same way but to allow for real values of the phase weight  $w_\phi$ .

The third option is to calculate the incoming excitation as  $(r^*w_r, \phi + w_\phi)$  with  $w_\phi$  either restricted to integer values or allowed to take real values.

It is clear that for any non-zero weight  $(w_r, w_\phi)$ , and any complex value  $(r, \phi)$ , there exists a weight  $(w_r', w_\phi')$  so that  $(r^*w_r, \phi w_\phi) = (r^*w_r', \phi + w_\phi')$ . In other words, the first and the third options are equivalent.

**Level 2: excitation.** The choice to be made here regards the way in which the excitation is calculated from the incoming excitations modified by the weights.

There are at least two different options. The first option is to add the incoming values as complex numbers. This option allows the transfer of information from the phase channel to the radius channel and/or vice versa.

The other option is to add the incoming values separately on the radius channel and phase channel. This option keeps the radius and phase channel separated.

**Level 3: the transfer function of a neuron.** The choice to be made regards the way the output of a neuron is calculated from its excitation.

There are two options here. The first possibility is to use an analytic complex function of a complex variable. However, Liouville's theorem states that "If the complex function  $F(*)$  is analytic and bounded everywhere in the complex plane, then  $F(*)$  is a constant". For gradient descent purposes, the activation function was thought to be necessarily analytic. Therefore, if the activation function is to be not-constant, it will be unbounded. Some attempts to extend backpropagation to the complex plane use an analytic, therefore unbounded activation function.

The second possibility is to use two independent real functions of a single real variable (radius/phase or real/imaginary). This would allow for bounded and differentiable radius/phase or real/imaginary parts of the activation function.

The actual choices are:

### **Level 1: weights.**

At this level, the choice is to alter separately the radius and phase values with radius and phase weights. Both the radius and phase weights are real to allow for gradient descent to be used by calculating and using derivatives. The finding of the appropriate weights is left entirely to the training algorithm.

There are two points to be made in connection to the physical meaning of the weights.

- i) By ensuring the correct output at the end of the training, the training algorithm should implicitly ensure the weights in the net are appropriate.
- ii) In this situation, there is no need for a correct physical meaning of the transformation brought by each and every weight as long as the output is the correct one and has therefore a correct physical interpretation.

### **Level 2: excitation.**

The choice is to add the incoming values as complex numbers. This method of calculating the excitation does not have a direct physical interpretation in signal space. That is, if all the incoming excitations were known vectors in signal space, the total excitation would not correspond to the sum vector of the incoming excitations. In other words, a neuron does not perform a vectorial sum in signal space of its inputs. One's objections regarding this lack of direct physical interpretation can be argued against along the same ideas as above. It is not important to have a direct physical interpretation for each and every excitation value in the net as long as the input and the output of the network i.e. at a global level, are the correct ones. Even in the case of the real brain, it is unreasonable to assume that a logical solution of a problem can be found by the brain only if all excitation values of the neurons in the brain can be put into some correspondence with some features of the problem.

The possibility to transfer the information from the radius channel to the phase channel and vice-versa is seen as an advantage not as a drawback. Thus, the training should ensure that this inter-channel communication is exploited only if is



needed. If at a given moment there is a need for taking into consideration phase values when calculating some radius value, or vice-versa, the possibility is there and it is the role of the training algorithm to ensure the right amount of information is passed from one channel to the other. On the other hand, if there is no need for this cross-channel communication, this transfer is not bound to appear and again, the training algorithm should take care of this.

### **Level 3: the transfer function of a neuron.**

Two independent real functions of a real variable have been chosen. This ensures the possibility of having two well defined, differentiable and monotonic function which are very convenient for backpropagation. Two standard logistic functions can be used or the identity function can be used for the phase channel.

#### **7.4.3.3. Detailed description of the processing performed by the net.**

In the chosen approach, the excitation is calculated in such a way that a neuron performs a sum of the complex numbers coming through the links. The complex numbers are kept in radius/phase form. A complex weight (radius and phase) performs a rotation (the phase will affect the angle) and a scaling (the radius will affect the magnitude) in the Argand plane (the complex plane).

$$\begin{cases} \text{out}^{ra} = \sigma_{ra}(ex^{ra}) \\ \text{out}^p = \sigma_p(ex^p) \end{cases}$$

$$\begin{cases} ex_k^x = \sum_{j=1}^n ex_j^x = \sum_{j=1}^n w_{jk}^{ra} r_j \cos(w_{jk}^p \theta_j) \\ ex_k^y = \sum_{j=1}^n ex_j^y = \sum_{j=1}^n w_{jk}^{ra} r_j \sin(w_{jk}^p \theta_j) \end{cases}$$

$$\begin{cases} ex_k^{ra} = \sqrt{(ex_k^x)^2 + (ex_k^y)^2} \\ ex_k^p = \arctg\left(\frac{ex_k^y}{ex_k^x}\right) \end{cases}$$

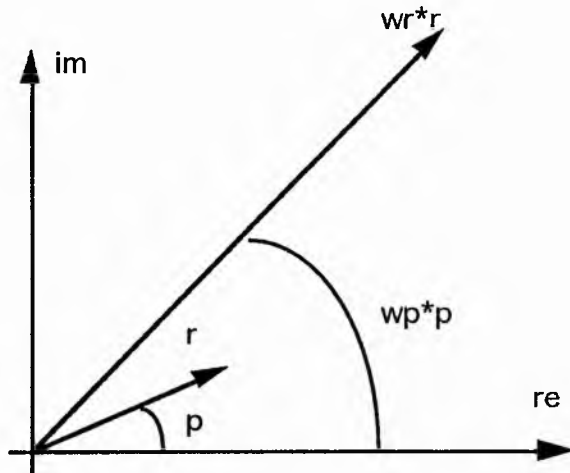


Fig. 1. The effect of a weight (radius =  $w_r$  and phase =  $w_p$ ) upon a  $(r, p)$  pair.

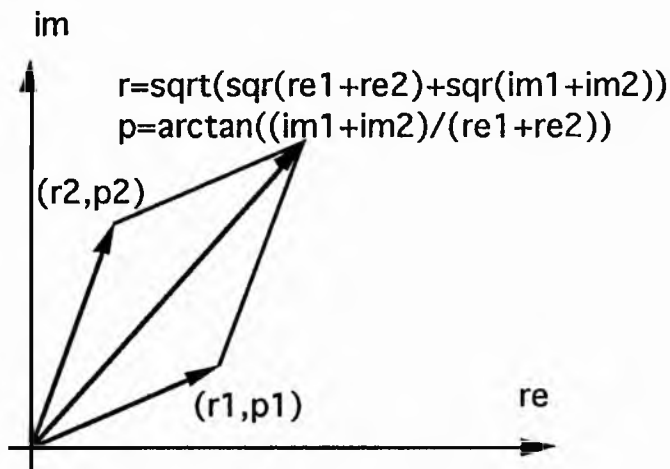


Fig. 2. What a neuron (with two incoming links) does.  $(r_1, p_1)$  and  $(r_2, p_2)$  are the incoming excitation i.e. each is already affected by the weights as in fig.1.

At the level of the neuron, the radius and phase parts are kept apart by using two different activation functions. Each of these functions is a real function of a real variable.

Because the inter-channel communication is possible at the level of calculating the excitation, such a network is not equivalent to a pair of real networks.

## 7.5 Training the complex network

### 7.5.1. The complex backpropagation algorithm.

Taking into consideration the choices made, the idea of backpropagating the error can be applied and the weight changes can be calculated as follows. The detailed derivation of these equations is presented in appendix 3.

Two different activation functions are used for the radius and phase channel:

$$\begin{cases} \text{out}^{ra} = \sigma_{ra}(ex^{ra}) \\ \text{out}^p = \sigma_p(ex^p) \end{cases}$$

The excitation of a neuron is calculated as the complex sum of the incoming excitations. Each weight has a radial component and a phase component and both of them multiply the radial and phase output of the unit on the previous layer respectively. Firstly, the real and imaginary components of the excitation of a neuron are calculated:

$$\begin{cases} ex_k^x = \sum_{j=1}^n ex_j^x = \sum_{j=1}^n w_{jk}^{ra} r_j \cos(w_{jk}^p \theta_j) \\ ex_k^y = \sum_{j=1}^n ex_j^y = \sum_{j=1}^n w_{jk}^{ra} r_j \sin(w_{jk}^p \theta_j) \end{cases}$$

and using these values, the radial and phase components of the excitation are calculated:

$$\begin{cases} ex_k^{ra} = \sqrt{(ex_k^x)^2 + (ex_k^y)^2} \\ ex_k^p = \arctg\left(\frac{ex_k^y}{ex_k^x}\right) \end{cases}$$

In these conditions, the weight change can be calculated so that the trajectory of the training in the error weight space follows the gradient of the error weight surface [Appendix 3; Weir, 1995]:

$$\begin{cases} \Delta w_{jk}^{ra} = -\varepsilon_{ra} \frac{\partial E}{\partial w_{jk}^{ra}} \\ \Delta w_{jk}^p = -\varepsilon_p \frac{\partial E}{\partial w_{jk}^p} \end{cases}$$

The partial derivatives of the error with respect to the radial and phase weights can be calculated as:

$$\begin{cases} \frac{\partial E}{\partial w_{jk}^{ra}} = \delta_k^{ra} r_j \cos(w_{jk}^p \theta_j - ex_k^p) + \delta_k^p \frac{r_j}{ex_k^{ra}} \sin(w_{jk}^p \theta_j - ex_k^p) \\ \frac{\partial E}{\partial w_{jk}^p} = \delta_k^{ra} r_j w_{jk}^{ra} \theta_j \sin(-w_{jk}^p \theta_j + ex_k^p) + \delta_k^p \frac{r_j}{ex_k^{ra}} w_{jk}^{ra} \cos(w_{jk}^p \theta_j - ex_k^p) \end{cases}$$

where the delta values are calculated as follows for the output units and hidden units respectively:

$$\begin{cases} \delta_k^{ra} = (out_k^{ra} - targ_k^{ra}) \sigma_{ra}' \\ \delta_k^p = (out_k^p - targ_k^p) \sigma_p' \end{cases}$$

$$\begin{cases} \delta_k^{ra} = \sigma_{ra}' \sum_k \left[ \delta_k^{ra} w_{jk}^{ra} \cos(w_{jk}^p \theta_j - ex_k^p) + \delta_k^p \frac{w_{jk}^{ra}}{ex_k^{ra}} \sin(w_{jk}^p \theta_j - ex_k^p) \right] \\ \delta_k^p = \sigma_p' \sum_k \left[ \delta_k^{ra} w_{jk}^{ra} r_j w_{jk}^p \sin(-w_{jk}^p \theta_j + ex_k^p) + \delta_k^p \frac{w_{jk}^{ra}}{ex_k^{ra}} r_j w_{jk}^p \cos(w_{jk}^p \theta_j - ex_k^p) \right] \end{cases}$$

Although much more complicated than the equations of the standard backpropagation, these equations illustrate the same idea: that the error (delta) values for any neuron can be calculated from the delta values of the neurons on the next layer and the weight values.

## 7.6 Relation to other work

The main approaches to using a complex network will be presented in the following.

A least-mean-square (LMS) adaptive algorithm for complex signals is presented in [Widrow, 1975]. In this algorithm, the weights are complex numbers expressed in real and imaginary form. The only processing at the neuron level is summing the excitations coming through the links. Therefore, this algorithm can be applied only to linear (one layer) networks.

A first generalisation of the backpropagation algorithm for multilayer networks to networks using complex weights and non-linear activation functions is given in [Leung, 1991]. In this approach, the weights are complex numbers in the real/imaginary form. The activation values from the previous layer are multiplied

with the weights as complex numbers. Therefore, the real weight will affect the imaginary part of the excitation coming through that link and vice-versa.

The activation function is a complex function of a complex variable. However, the function maps a real number to a real number and a complex number to a complex number. The singularities of the sigmoid activation function are avoided by mapping the input onto some suitable region of the complex plane.

$$f(A_j) = f(x + iy) = \frac{1}{1 + \exp(-A_j)} = \frac{1}{1 + \exp(-(x + iy))}$$

In [Leung, 1991], the network is used for a very simple classification problem ( 2 input vectors to be classified by a 4-2-4 architecture). The purpose of the experiment is to show that the algorithm works. The generalisation issues are not discussed and no comparison with the real networks is attempted.

An approach similar to that of Leung is presented in [Kim, 1990]. The weights are used in real and imaginary terms and the same complex activation function is used.

Although the motivation given for using a complex network is signal processing in frequency domain, the functioning of the complex network is illustrated on a complex generalisation of the exclusive OR problem.

Clarke in [Clarke, 1990] discusses a complex network with just one neuron and different activation functions. A single complex neuron with an activation function like

$$f(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \text{ and}$$

$$f(z) = \frac{z - \alpha}{1 - \alpha^* z}$$

where  $\alpha$  is a complex constant of magnitude less than 1 and  $*$  denotes the complex conjugation.

The issues raised by an analytic complex activation function<sup>1</sup> are discussed and the unavoidability of singularities is emphasised. The possibility of a single neuron to implement complicated functions is presented as the motivation for further study of complex networks. However, no attempt of studying more complicated networks is done in this paper.

In 1989, Birx and Pipenberg published a paper on signal processing for defect discrimination [Birx, 1989]. In this paper, they try various approaches to non-destructive defect detection using neural networks. After trying several network structures, Birx and Pipenberg are forced to use a complex network which allows them to incorporate both phase and amplitude information. Their complex network uses weights in real/imaginary form and two sigmoid functions, for the real and imaginary components. The paper concentrates on the practical aspects of the problem.

Little et.al discuss a possible implementation of a neural network in an optical device [Little, 1990]. In their network, the weights are real and imaginary but "the non-linear operation in a neuron is a function only of the optical intensity at the neuron". In other words, the weights are complex but all the other values in the network (excitations, inputs, outputs, etc.) are real. Here, as in [Birx, 1989], the choice is dictated by practical considerations: "This mode of operation is clearly appropriate when the non-linear operation is implemented optoelectronically using direct detection followed by electronic thresholding, and it may be applicable for many all-optical non-linear processes". The paper presents an example in which such a complex network is used to train an XOR problem (the training is simulated) with a 2-2-1 architecture.

In 1991, Benvenuto pointed out that the activation function of a complex neuron need not be analytic for the gradient descent to be applied [Benvenuto, 1991]. This avoids a consequence of Liouville's theorem: "If the complex function  $F(*)$  is analytic and bounded everywhere in the complex plane, then  $F(*)$  is a constant". The immediate consequence of this theorem is that if the function is analytic then it is either a constant (which is absolutely useless as an activation function for a neural network) or it is not bounded (i.e. it has singularities). Benvenuto proposes an

---

<sup>1</sup>A function of a complex variable  $z$  is analytic at  $z_0$  if there is a neighbourhood  $U$  of  $z_0$  such that the function is differentiable at each point of  $U$ .

activation function which squashes both the real and imaginary components of the complex excitation and takes the squashed values as the real and imaginary parts of the complex activation value:

$$SGM(Z) = sgm(z_R) + js gm(z_I)$$

where the excitation of the neuron is calculated as:

$$S_n^{(l)} = \sum_{m=0}^{N_{l-1}} W_{nm}^{(l)} X_m^{(l-1)}$$

where W and X are complex weights and respective complex activation values in real and imaginary form.

The sigmoid presented in this paper, although it is not analytic, allows the extension of the BP algorithm to the complex plane and is bounded everywhere in the complex plane.

No motivation for the use of a complex network is given in this paper and no experiments are presented. These lacks are filled in [Benvenuto, 1992]. In this paper, two approaches to dealing with complex data in neural networks are presented:

- a) to use a classical network where the complex input and output signals of the network are replaced by pairs of independent real-valued signals;
- b) to use a complex network where the neurons, the functioning and the training of the networks have been redefined to work with complex values.

The second approach is chosen and used within the same framework as in [Benvenuto, 1991]. The complex network is compared with a real network with twice as many units in an application dealing with equalising a data signal distorted by non-linear channels characteristics to PSK/TDMA satellite transmission systems. The choice of such a highly specialised application is justified to the authors because "the extension of some classical problems, like the XOR problem, to the complex plane seems to be actually useless".

Nitta in [Nitta, 1993] presents a similar complex network. This network uses real and imaginary weights and two sigmoids, for the real and imaginary excitations exactly as in [Benvenuto, 1992] and [Benvenuto, 1991]. This paper however, presents some simple experiments which show that a complex network trained with

certain transformation on a line (in the unit disc in the complex plane) manages to exhibit the same behaviour on a larger domain (included in the same unit disc). For comparison, a standard network with twice the number of units fails to do so.

The complex network presented in this thesis differs from the existing techniques in its motivation, design and implementation.

The motivation of the approach presented is to embed in the weight state of the network more than the information contained in a number of samples, if that information is available. This would allow the network to yield an I/O behaviour better than the generalisation of a standard network trained with samples. This motivation comes from a signal processing point of view but is not limited to the requirements of any particular application.

The design of the network presented is based on the general requirements of the approach. The network uses radial and phase weights and different activation functions for the radial and phase parts. This comes along the lines drawn by Benvenuto which showed that the activation function needs not be analytic for an appropriate backpropagation algorithm to be derived. However, the approach presented in this thesis shows that an appropriate backpropagation algorithm can be used even if the complex activation function is constructed in radius and phase terms even with different (sigmoid for the radius and linear for the phase) real activation functions. The network used in the experiments was implemented with sigmoidal activation function for the radial part and a linear activation function for the phase part. The choice of using radial and phase weights for the complex network and the activation functions was determined by the motivation given and is specific to this approach.

The same can be said about the pre-processing and the interpretation of the values applied to the network. The approach presented in [Birx, 1989] uses a Fast Fourier Transform (FFT) to pre-process the data and to obtain magnitude and phase information. However, in that case, the real and imaginary values coming from the FFT are applied directly to the network. A real/imaginary pair will be applied to one input neuron. In the approach presented in this thesis, a real-imaginary pair will be applied to two neurons. Each of these two neurons will be fed with radius information which corresponds to the real or imaginary value (with the values suitably mapped onto some convenient interval) and phase information which corresponds to the frequency information (with the values suitably mapped onto some convenient interval).



## **7.7. Conclusions**

### **7.7.1 Main ideas in CBP:**

1. Generalisation of a neural network is given by the behaviour in between data points. Good generalisation means obtaining the desired behaviour in between data points as defined by a known underlying function or problem specific expectations.

2. The classic approach to modelling a phenomenon with a neural network is:

a) Take samples from the input and the output of the black box representing the given phenomenon.

b) Train a network which (hopefully) gives the correct output for all the input samples

c) Hope that the network will give appropriate outputs for the inputs which are not in the training set.

3. The signal approach to modelling a phenomenon with a neural network is:

a) Treat the phenomenon in its entirety by taking into consideration the input and output signals and not only samples of them.

b) Obtain a parametric representation of these input and output signals. Ideally, these parametric representations contain all the information in the original signals in a finite form. In practice, one cannot have a universal parametric representation but it is believed that in most cases, the information contained in such parametric representation will be more than the information contained in a set of samples. This is because a set of samples allows an arbitrary variation between the samples whereas a parametric representation imposes some sort of default shape which perhaps can be conveniently chosen.

c) Train a network to associate the parametric representation of the input signal to the parametric representation of the output signal.

d) Since these parametric representations contain more information (regarding the shape of the signals) than an arbitrary set of samples, the underlying phenomenon will be more faithfully modelled i.e. the generalisation should be better than the generalisation given by the average network trained with samples. If the parametric representation of the signal is an exact one, the generalisation yielded by this

approach is the ideal generalisation. Although in the strict sense of the word as defined in 1, one does not have generalisation anymore because there are no more samples, the term ideal generalisation was used to describe the situation in which all the information present in the original signal is stored in the network.

### **7.7.2 Relation between the parallel approach, 'simple shaping' and 'rotation and scaling'.**

Any technique associating parametric representations instead of instantaneous values feeds in some implicit information regarding the shape. Thus, this approach ensures some control over the shape of the I/O function although not by itself but because of the parametric representation. In other words, this feature is not specific to this approach but characterises any approach using a parametric representation association instead of a sample association.

Within the general framework of the parametric representation association, the signal space approach has been chosen as a possibility. If the parametric representation is to be chosen as defined by the signal space approach (a Fourier representation for instance), then the rotation and scaling are the operations necessary to transform a representation of a signal into a representation of a different signal. As this is necessary for associating the input signal to the output signal, these operations must have some correspondents in the operations performed in the complex network.

Thus, the rotation and scaling give some suggestions regarding various choices possible in the implementation of this approach using a complex network.

### **7.7.3 Caution regarding the usage of the complex backpropagation.**

Sometimes, the only available information about the phenomenon to be modelled is in a form of samples. In this case, in principle, the signal approach requires a function to be fitted through the data points first (an input function and an output one). Subsequently, these functions will be treated as the underlying phenomenon.

This function fitting step is not trivial. In some sense, the whole difficulty of the problem is concentrated here because by fitting a function through a number of given samples, one implicitly defines the generalisation. If wrong behaviour is induced in the model at this stage, there is little the signal approach can do afterwards.

However, the Fourier approach used as an example might seem to be a safe option. The approach presented reduces to a Fourier series decomposition if the function is periodical in the time domain (or can be approximated as such). This decomposition is followed by an association (performed by the network) of the parameters characteristic to this decomposition.

If the function is not periodical, the representation in the Fourier space will not be discrete anymore. However, it is believed that the association of signals in the frequency domain as opposed to associating signals in time domain can bring improvements as suggested in this thesis.

# CHAPTER 8

## Experiments with the Complex Backpropagation (CBP)

### 8.1 Introduction

In this chapter, several experiments designed to test the complex backpropagation approach are presented.

In section §8.2, the objectives of these experiments are presented and their general framework is described. Section §8.3 describes very briefly the experiments and their results. Section §8.4 presents the calculation of the error limit to be used in the complex backpropagation experiments. Section §8.5 presents the details of the generalisation experiments and a comparison between complex backpropagation and standard backpropagation. The conclusions are presented in section §8.6.

The details of the experiments investigating the training properties of the complex backpropagation (aimed mostly at the reader interested in reproducing the experiments) are given in appendix 4.

### 8.2 The objectives of the experiments

1. The first objective of the experiments is to test the training algorithm i.e. the ability of the complex training to converge towards a solution weight state satisfying the I/O patterns. Such a convergence would prove the correctness<sup>1</sup> of the algorithm and its implementation.

In order to test this without having to be concerned with the existence of the solution and with the capacity of the architecture to solve a randomly chosen I/O problem, these experiments were set-up in the following way. Firstly, a random weight state for the given architecture was generated. Using this weight state and a set of random input patterns, some output values were obtained. Then, the network

---

<sup>1</sup>The term correctness is used to designate the property of the algorithm of being able to yield the desired result (a low error weight state) in most cases. The experiments do not intend to test or prove the formal correctness of the algorithm.

is initialised with a new random weight state and trained with the I/O patterns obtained as above. This ensures that a solution weight state does exist for the given architecture and the given training set and furthermore, if the training finds a solution, this solution can be compared with the "original" weight state i.e. the weight state used to generate the I/O patterns in the training set.

This test should be performed at least with an architecture without hidden units and with an architecture with hidden units. This is necessary because the formulae for the delta values have different expressions for the last layer of neurons and a hidden layer.

Two different types of convergence are sought. The aim of the first type is to find the original weight state i.e. the one used to generate the I/O patterns. In experiments aimed at this target, the architecture of the trained network is the same with the architecture used to generate the I/O state and the training is performed with an extremely small error limit ( $10^{-8}$  maximum absolute difference between a target radius or phase value and its actual correspondent ) to force the net to find the precise weight state. This very small error limit at the level of the output corresponds to an error limit of about  $10^{-3}$ - $10^{-2}$  at the weight level.

The aim of the second type of convergence is the usual aim of a neural network training i.e. to bring the error limit below a reasonably small error limit which would subsequently allow the use of the net. This second type of experiment is performed with various architectures in order to test the ability of the complex network/training algorithm to find a solution (be it even only a partial solution) when the architecture is different from the "ideal" one. The ideal architecture is the minimal architecture for which there exists a weight state with non zero weights able to yield zero error. If the architecture has fewer hidden units than the ideal one, one expects the network to find a partial solution which gives the correct output only for some of the input patterns only (i.e. the error for some input patterns is below the error limit) or -more frequently- gives a relatively small error for all input patterns but this error is above the required error limit. If the architecture has more hidden units than the ideal one, one expects the network to find a solution which yields the desired output within the error limit for all input patterns in the training set. The reason for this is that for too rich an architecture a zero error weight state always exists. Such a weight state is, for instance, the weight state used in generating the I/O patterns for the ideal architecture, completed with zero weights for all the supplementary links. However, it is expected that the network never find

this type of solution and give instead an alternate solution perhaps exhibiting some overfitting behaviour.

2. The second objective of the experiments is to test the abilities of the training algorithm to find a solution weight state for a randomly generated I/O problem.

This should test the versatility of the training algorithm and that of various architectures. This experiment must be seen in the light of the architectural issues discussed in chapter 2. If a training session fails, whose responsibility is? Is it training algorithm's fault because it does not explore the weight space in an efficient manner or architecture's fault because is too simple for the given I/O problem? Unfortunately, this question cannot be answered in the general case not even for standard networks.

A very small error limit ( $10^{-8}$  maximum absolute difference between a target radius or phase value and its actual correspondent) should be used in this case as well to ensure that the solution is a genuine one and not only a local minimum which happens to have a small error.

3. Finally, the third objective of the experiments is to test the generalisation abilities of the network in the given framework. This experiment will present the method for associating signals (including the pre-processing needed) and how the results obtained with the complex backpropagation compare with the results obtained with the standard multilayer perceptron.

### 8.3. Short description of the experiments. Results in brief

#### 8.3.1. Testing the training algorithm.

##### Experiment 1

The first experiment involves just one link between two neurons in the conditions in which no bias is used. The architecture is presented in fig. 1. As explained, a random weight state and two randomly chosen input patterns are used to generate a training set containing two complex I/O patterns.

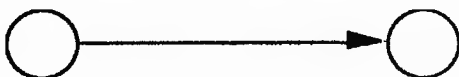


Fig. 1 A one-link architecture, no bias.

In three out of five training sessions, the net converged to the original weight state fairly quickly. In the other two sessions the net failed to converge altogether.

### Experiment 2

The second experiment used a 1-1 architecture with bias trained with the same two patterns as in experiment 1. The architecture is presented in fig. 2.

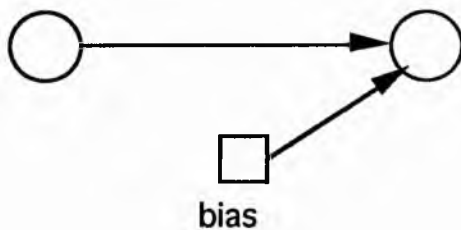


Fig. 2 A one-link architecture with bias.

In all five training sessions, the network converged to the original weight state. In general, the convergence was much slower than the successful training sessions of the architecture 1-1 without bias.

### Experiment 3

This experiment used an architecture with two links and without bias as in fig. 3



Fig. 3. A two-link architecture without bias

As this architecture is fundamentally different from the one used to generate the patterns, the sought outcome is finding a weight state with a low error limit. In this case, no comparison between the original weight state and a potential solution is possible.

The training of this architecture succeeded in two out of five trials.

### Experiment 4

This experiment used an architecture with two links and bias as in fig. 4. This architecture was trained with four patterns generated with the same 11 architecture with bias.

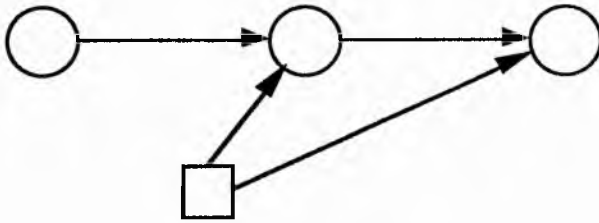


Fig. 4. A two-link architecture with bias

All five training sessions were successful. In each case, a weight state yielding a small error was found.

### Experiment 5

This experiment used an architecture with four links and bias as in fig. 5

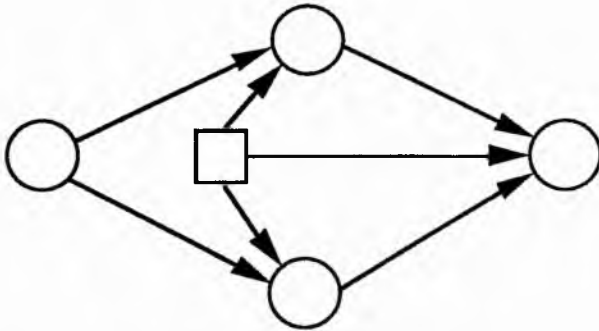


Fig. 5. A two-link architecture with bias

This architecture was trained with a pattern set containing four complex patterns.

As the architecture used during the training is the one used to generate the patterns, the sought outcome is finding the original weight state. Five training sessions were performed and all of them yielded solution weight states different from the original weight state but characterised by a low error.

### **8.3.2. Testing the training algorithm with a random I/O problem.**

#### Experiment 1

A 1-2-1 architecture with bias was trained with 4 random patterns. The patterns are formed with random radius value in the range (0.5, 1) and random phase values in the range (-1,1).

Five different initial random weight states were tried. All five training session were successful in the sense that the error at the end of the training was very low, but all



of them needed a very large number of iterations. This seems to indicate that the algorithm is indeed able to yield low error weight states.

## Experiment 2

A 1-3-1 architecture with bias is trained with 4 random patterns. The same patterns as for the 121 architecture were used.

Once again, five different initial random weight states were used. All five training sessions were successful but all of them needed a very large number of iterations although in four out of five cases the number of iterations needed was less than for the correspondent training session of the 121 architecture. This seems to indicate that, as for the standard multilayer perceptron, a small increase in the number of hidden units (free parameters) brings an increase in capabilities.

### **8.3.3. Testing the generalisation**

The simple example of a toy-problem was chosen to illustrate the approach and the processing steps involved. Subsequently, a more complicated problem is used to further test the approach and compare it with the standard one.

The aim of the first generalisation experiment is to associate an input signal to an output one i.e. to implement a single signal association. The chosen input signal is:

$$f(t) = 1 + \cos(2\pi f_0 t) - 2\sin(2\pi f_0 t) + 3\cos(2\pi 2f_0 t) + 4\sin(2\pi 2f_0 t)$$

and the chosen output signal is:

$$f(t) = 2 + 4\cos(2\pi f_0 t) - 3\sin(2\pi f_0 t) + 1\cos(2\pi 2f_0 t) + \sin(2\pi 2f_0 t)$$

The fundamental frequency  $f_0$  is taken so that a period of the fundamental has the length 1. Both the input and the output signal are given through 16 samples uniformly spaced in the interval (0,1).

Firstly, the task is solved using a network with complex weights. Then, a standard backpropagation network is used to accomplish the same task and the results are compared from the point of view of generalisation. This generalisation comparison is performed by comparing the values yielded by the standard and signal space approach with those of the desired output function in 128 points equally spaced in the same interval (0,1).

This experiment was successful, showing that the complex network can implement the desired signal association with lower error. This experiment is presented in detail in section 8.5.1.1.

A more difficult task was then presented to the networks. This time, the task is to implement an amplitude independent signal association i.e. to associate a given input signal  $s_1$  to a given output signal  $s_2$  on various amplitude scales. Given two real numbers  $k_1$  and  $k_2$ , the training set will contain the association of  $k_1s_1(t)$  to  $k_1s_2(t)$  and  $k_2s_1(t)$  to  $k_2s_2(t)$ . The purpose is to determine the networks to associate any signal  $k_3s_1$  to the signal  $k_3s_2$  and this is tested after training by presenting the network with an input signal  $k_3s_1(t)$  and comparing it with the desired output signal  $k_3s_2(t)$ . This experiment is presented in detail in section 8.5.1.2.

This experiment might seem an overly simple test because it asks a one-layer network to implement a linear transformation. This is because a linear perceptron trained with only few patterns (for instance two patterns for a 2D input space) would give the right generalisation answer for any number of supplementary patterns as long as the desired I/O relationship obeys the linear relationship of the trained hyperplane. However, the complex backpropagation network is not a linear perceptron. The transfer function implemented by the complex network is not linear and there is a great diversity of functions which can be modelled (on a finite interval) by the complex network. Furthermore, the required linear transformation is performed on non-linear functions.

A comparison between the results obtained with the standard backpropagation and the results obtained with the complex backpropagation was performed. This comparison showed that the complex network offers better possibilities for this type of problems yielding lower generalisation errors.

#### **8.4 Calculating the error limit**

Some of the experiments aim at retrieving a given weight state from a set of I/O patterns generated with this given weight state. An immediate question arises: how large an error limit must be used during the training in order to ensure a meaningful comparison of the weight state found during the training with the original weight state used to generate the patterns? In other words, what is the relationship between the errors at the weight level and the errors at the output level or alternatively, how do the errors in weights propagate to the output level.

This problem reduces to the following problem: given the errors of a certain set of quantities, to determine the error of a given function of these quantities. This is a typical problem in the theory of errors. A brief justification for the formulae used will be given in the following. More details can be found in [Demidovich, 1981].

Suppose a differentiable function

$$u = f(x_1, x_2, \dots, x_n) \quad (1)$$

is given. Let  $|\Delta x_i|$   $i = 1, 2, \dots, n$  be the absolute errors of the arguments of the function. Then the absolute error of the function is:

$$|\Delta u| = |f(x_1 + \Delta x_1, x_2 + \Delta x_2, \dots, x_n + \Delta x_n) - f(x_1, x_2, \dots, x_n)| \quad (2)$$

In practice the absolute errors of the arguments  $|\Delta x_i|$  are small quantities and their products, squares and higher powers can be ignored. In these conditions:

$$|\Delta u| \approx |df(x_1, x_2, \dots, x_n)| = \left| \sum_{i=1}^n \frac{\partial f}{\partial x_i} \Delta x_i \right| \leq \sum_{i=1}^n \left| \frac{\partial f}{\partial x_i} \right| |\Delta x_i| \quad (3)$$

From this expression, denoting by  $\Delta x_i$  the limiting absolute errors of the arguments  $x_i$  and by  $\Delta u$  the limiting absolute error of the function  $u$ , one obtains for small  $\Delta x_i$ :

$$\Delta u = \sum_{i=1}^n \left| \frac{\partial f}{\partial x_i} \right| \Delta x_i \quad (4)$$

This formula allows the calculating of the error of the function when the error of the arguments is known. For our problem, this formula would allow the calculation of the output error determined by weight errors if we knew the gradient of the function  $f$ . In our case the function  $f$  is:

$$out = f(w^r, w^p) \quad (5)$$

where the input is given. Function  $f$  has two components, one for radius and one for phase. Note that function  $f$  is different in the neighbourhood of different input points.

Our problem is reversed though: what error must we ask the output to satisfy in order to obtain a desired error in the weights. In other words: how does the error propagate backwards from the output to the weights? Or even: given an absolute

error for the output, what is the absolute error at the weight level (the absolute error of a particular weight) which caused the given output error?

This problem is mathematically indeterminate since the error of the function can be ensured by establishing the errors of the arguments in different ways. According to [Demidovich, 1981], the simplest solution of the inverse problem is given by the so called *principle of equal effects*. According to this principle, it is assumed that all the partial differentials:

$$\frac{\partial f}{\partial x_i} \Delta x_i, i = 1, 2, \dots, n \quad (6)$$

contribute to the same extent at the total absolute error  $\Delta_u$  of the function

$$u = f(x_1, x_2, \dots, x_n) \quad (7)$$

If the magnitude of the absolute error  $\Delta_u$  is given, then from (4) and (6) it follows that:

$$\Delta_u = \sum_{i=1}^n \left| \frac{\partial f}{\partial x_i} \right| \Delta x_i = n \left| \frac{\partial f}{\partial x_i} \right| \Delta x_i \text{ and hence} \quad (8)$$

$$\Delta x_i = \frac{\Delta_u}{n \left| \frac{\partial f}{\partial x_i} \right|} \text{ for any } i. \quad (9)$$

This allows the calculation of the error at the weight level when the error at the output level and the gradient of  $f$  are known.

The gradient of  $f$  can be estimated using the perturbation method:

$$\frac{\partial f}{\partial x_i}(x_{i0}) \approx \frac{\Delta f}{\Delta x_i} = \frac{f(x_{i0}) - f(x_{i0} + \Delta x_i)}{\Delta x_i} \quad (10)$$

In order to estimate the gradient of this output-weight function, an input pair and a set of weights are chosen. The output of the network is calculated for the chosen inputs and weights. Then, the weights are perturbed with a small quantity, firstly the radius and then the phase. The new output values of the network are calculated after each perturbation. Using these values the gradient of  $f$  can be estimated.

The approximation of the gradient given by the perturbation method is valid only in a neighbourhood of the point  $x_0$  (i.e. for a given input pair). However, we are

interested in calculating only the order of magnitude of the error and therefore, the order of magnitude of the gradient will suffice. It is assumed that the gradient of the output-weight function does not change with more than an order of magnitude for the range of weights  $(-2, 2)$  considered.

Since the gradient of the function is different for different input pairs and in order to obtain a better estimate of the order of magnitude of the gradient, three input pairs have been used with the same perturbation 0.001 to calculate an average gradient of the output-weight function. The output values are presented in table 1.

	no perturb.	no perturb.	radius perturbation=0.001		phase perturbation=0.001	
	output radius	output phase	output radius	output phase	output radius	output phase
test 0	0.650389	0.012776	0.650407	0.01276	0.650385	0.012636
test 1	0.625304	-0.03586	0.625316	-0.03589	0.625281	-0.03624
test 2	0.616616	0.136995	0.616622	0.136978	0.616619	0.13707
			output difference		output difference	
			radius	phase	radius	phase
test 0			-1.8E-05	1.6E-05	4E-06	0.00014
test 1			-1.2E-05	3.4E-05	2.3E-05	0.00038
test 2			-6E-06	1.7E-05	-3E-06	-7.5E-05
		average	-1.2E-05	2.23E-05	8E-06	0.000148

Table 1. Estimating the gradient of the output-weight function using the perturbation method. The output values for unperturbed weight state and the 0.001 radius and phase perturbation are given.

Let us assume we are interested in obtaining the weight with an absolute error smaller than  $10^{-3}$  which would allow us to declare the weight states as being 'equal'. From (10) and the data in table 1 ( $\Delta x_i$  is the perturbation 0.001 and  $\Delta f$  is  $10^{-6}$  taken from the smallest value in the last row of the table), the gradient of the output-weight function with respect to the weight is estimated to be of the order of magnitude of  $10^{-3}$ :

$$\left| \frac{\partial f}{\partial x_i} \right| \approx \left| \frac{\Delta f}{\Delta x_i} \right| \approx \frac{10^{-6}}{10^{-3}} = 10^{-3} \quad (11)$$

From (9) one can estimate the order of the magnitude of the error at the output level to be  $10^{-6}$ . Indeed, substituting these values in (9) one obtains:

$$\Delta_{x_i} \approx \frac{\Delta_u}{\left| \frac{\partial f}{\partial x_i} \right|} \approx \frac{10^{-6}}{10^{-3}} = 10^{-3} \quad (12)$$

which is the order of magnitude we sought for the weights.

An output error limit of  $10^{-8}$  was taken to cater for the errors introduced by various approximations used and for eventual variations of the estimated gradient.

This value was confirmed post-factum by the differences between the trained weight states and the original weight state (those used to generate the I/O patterns). These differences were of the order  $10^{-2}$ - $10^{-3}$  which showed that the estimation above was correct.

## 8.5. Experimental details

The experiments performed can be divided in two large categories: experiments investigating the training properties and those investigating the generalisation properties of the complex network. The experiments investigating the training properties of the complex network have the main purpose to ensure that the complex backpropagation training algorithm was implemented correctly. Therefore, they constitute an important but somehow secondary part. On the other hand, the experiments investigating the generalisation properties of the complex network are in the focus of this chapter. For this reasons, this section will present in detail the experiments dealing with generalisation only. The details of the training experiments, aimed mainly at the reader who is interested in reproducing the results and/or further experimental investigations, are presented in appendix 4.

### 8.5.1. Testing the generalisation

The aim of the first generalisation experiment is to associate an input signal to an output one i.e. to implement a single signal association. First, the task is solved using a network with complex weights. Then, a standard backpropagation network

is used to accomplish the same task and the results are compared from the point of view of generalisation. Some brief remarks about training will also be made.

A more difficult task is then presented to the networks. This time, the task is to implement an amplitude independent signal association i.e. to associate a given input signal  $s_1$  to a given output signal  $s_2$  on various amplitude scales. Giving two real numbers  $k_1$  and  $k_2$ , the training set will contain the association of  $k_1s_1$  to  $k_1s_2$  and  $k_2s_1$  to  $k_2s_2$ . The purpose is to determine the networks to associate any signal  $k_3s_1$  to the signal  $k_3s_2$ . The success of this is tested after training by presenting the network with an input signal on an untrained amplitude scale  $k_3s_1$  and comparing it with the desired output signal  $k_3s_2$ . A comparison between the results obtained with the standard backpropagation and the results obtained with the complex backpropagation is then presented.

#### **8.5.1.1 A single signal association**

The chosen input signal is:

$$f(t) = 1 + \cos(2\pi f_0 t) - 2\sin(2\pi f_0 t) + 3\cos(2\pi 2f_0 t) + 4\sin(2\pi 2f_0 t)$$

and the chosen output signal is:

$$f(t) = 2 + 4\cos(2\pi f_0 t) - 3\sin(2\pi f_0 t) + 2\cos(2\pi 2f_0 t) + \sin(2\pi 2f_0 t)$$

The fundamental frequency  $f_0$  is taken so that a period of the fundamental has the length 1. The signals have been chosen so that i) they are not too simple in order for this example to be a non-trivial one and ii) they are not too complicated so that the results are intuitive and easy to interpret. The input signal is presented in fig. 6. The output signal is presented in fig. 7.

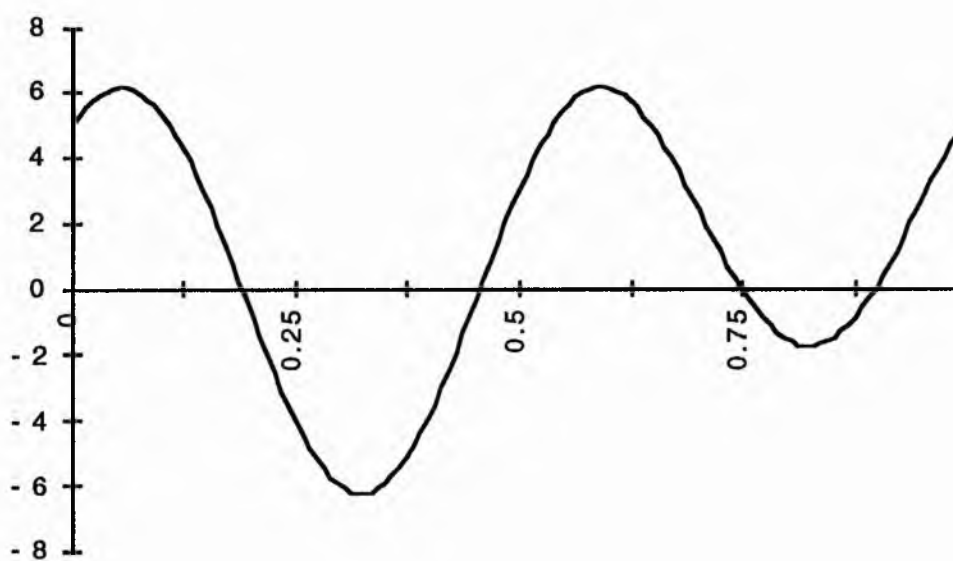


Fig. 6 The input signal

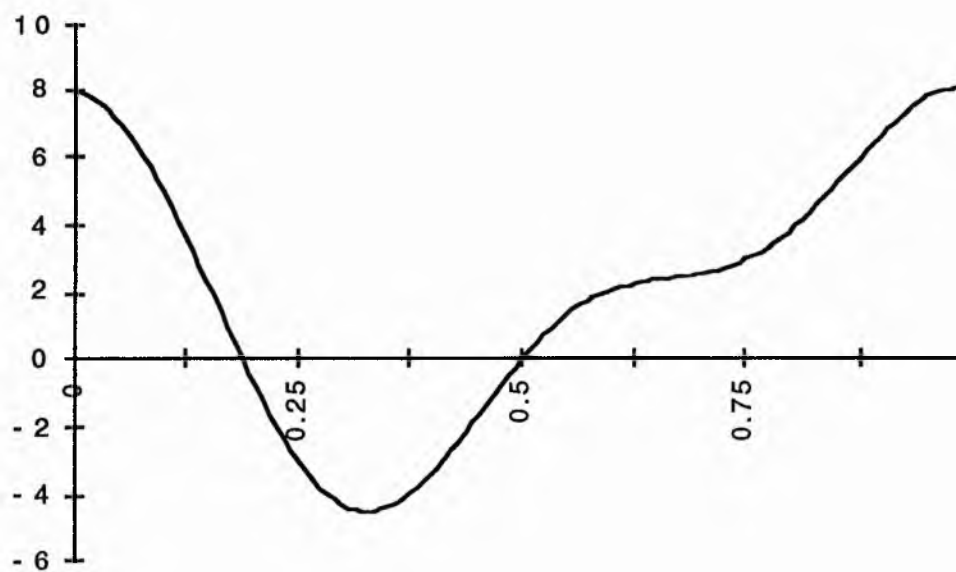


Fig. 7 The output signal.



### Standard backpropagation

The 16 samples from the input and output signals (presented in table 2) were mapped through a linear function to obtain the input/output values for the standard network presented in table 3.

no. of sample	time value	input signal	output signal
0	0	5	8
1	0.0625	6.10826	6.668788
2	0.125	4.292893	3.707107
3	0.1875	0.242031	0.051988
4	0.25	-4	-3
5	0.3125	-6.18019	-4.423693
6	0.375	-5.12132	-3.949747
7	0.4375	-1.396353	-2.136462
8	0.5	3	0
9	0.5625	5.791235	1.573853
10	0.625	5.707107	2.292893
11	0.6875	3.172182	2.533798
12	0.75	0	3
13	0.8125	-1.719305	4.181052
14	0.875	-0.87868	5.949747
15	0.9375	1.98214	7.550675

Table 2. 16 samples from the input and output signal.

training patterns		
input 1(time)	input 2	output
0	0.75	0.9
0.0625	0.805413	0.8334394
0.125	0.71464465	0.68535535
0.1875	0.51210155	0.5025994
0.25	0.3	0.35
0.3125	0.1909905	0.27881535
0.375	0.243934	0.30251265
0.4375	0.43018235	0.3931769
0.5	0.65	0.5
0.5625	0.78956175	0.57869265
0.625	0.78535535	0.61464465
0.6875	0.6586091	0.6266899
0.75	0.5	0.65
0.8125	0.41403475	0.7090526
0.875	0.456066	0.79748735
0.9375	0.599107	0.87753375

Table 3 I/O values for the standard network

These values were trained with a standard backpropagation algorithm and a 2-12-1 architecture. The standard network was trained with time values at one input neuron and correspondent samples of the input function at the other input neuron. The desired output value was taken to be the correspondent sample of the output function. For completeness, the network should have had another output neuron which would have given the time value associated with the output sample. However, in this case the output time value is always the same with the input time value by definition of the desired input and output signals and this unit was not introduced.

A training curve (the error as a function of the number of epochs) is given in fig. 8. The maximum absolute error after almost half a million epochs (447,000) was just above 0.01055.

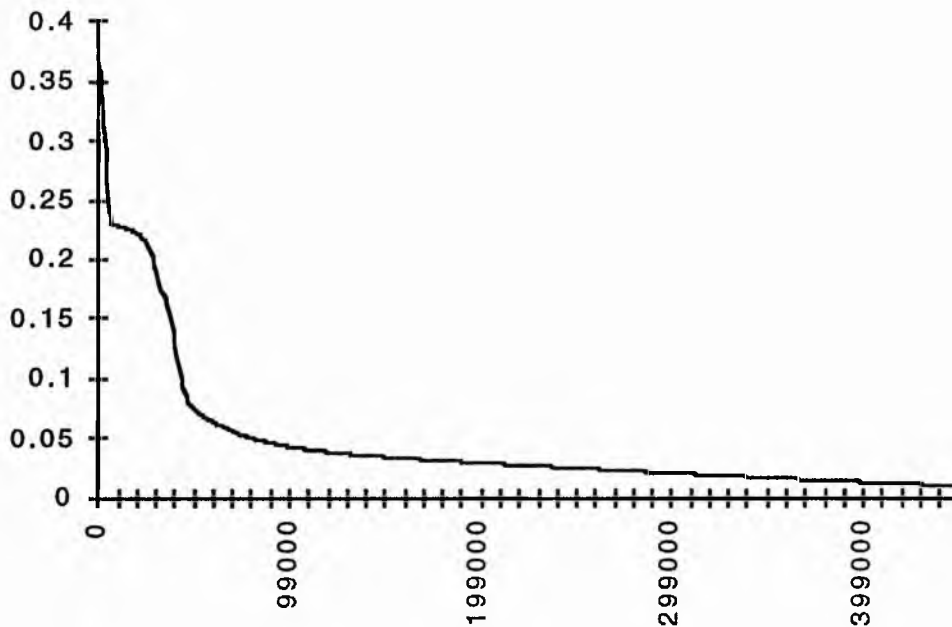


Fig. 8 The evolution of the maximum error per pattern set as a function of the number of epochs for the standard backpropagation network. The pattern set contains 16 patterns and the architecture is 2-12-1.

Fig. 9 presents the desired output function and the output of the standard network sampled in 128 time values between 0 and 1. Fig. 10 presents the same two functions on a smaller interval, between 0.75 and 1. It is now apparent that the output of the network trained with backpropagation does not fit exactly the desired output function. In this picture, it can be seen that the graphs of the two functions intersect only in a reduced number of points. Practically, the output of the network is close to the desired output only in the points which were in the training set (3 points between 0.75 and 1).

For our purpose, at this stage, it is sufficient to observe that the difference between the desired function and the output of the network is non-negligible for the scale chosen.

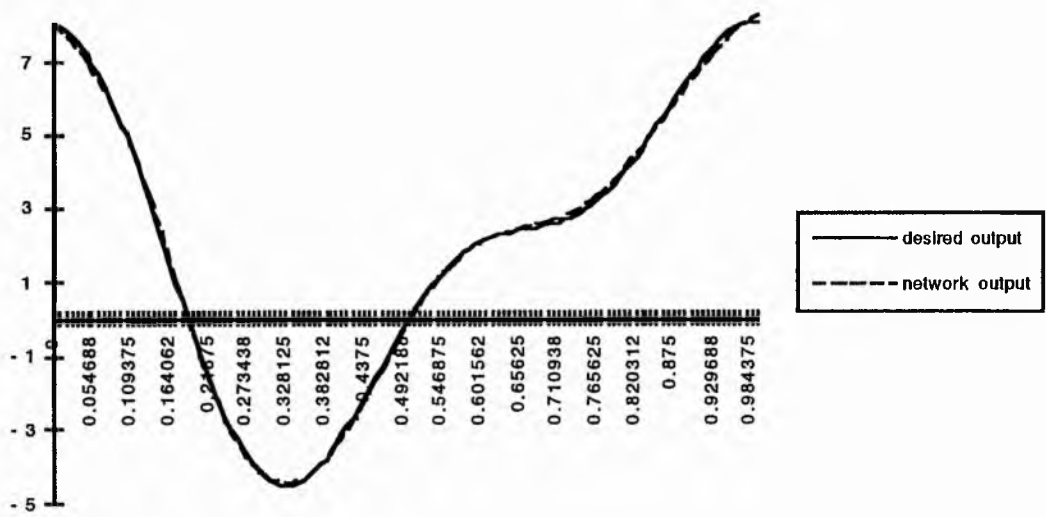


Fig. 9. The desired output function and the output of the network in 128 points between 0 and 1.

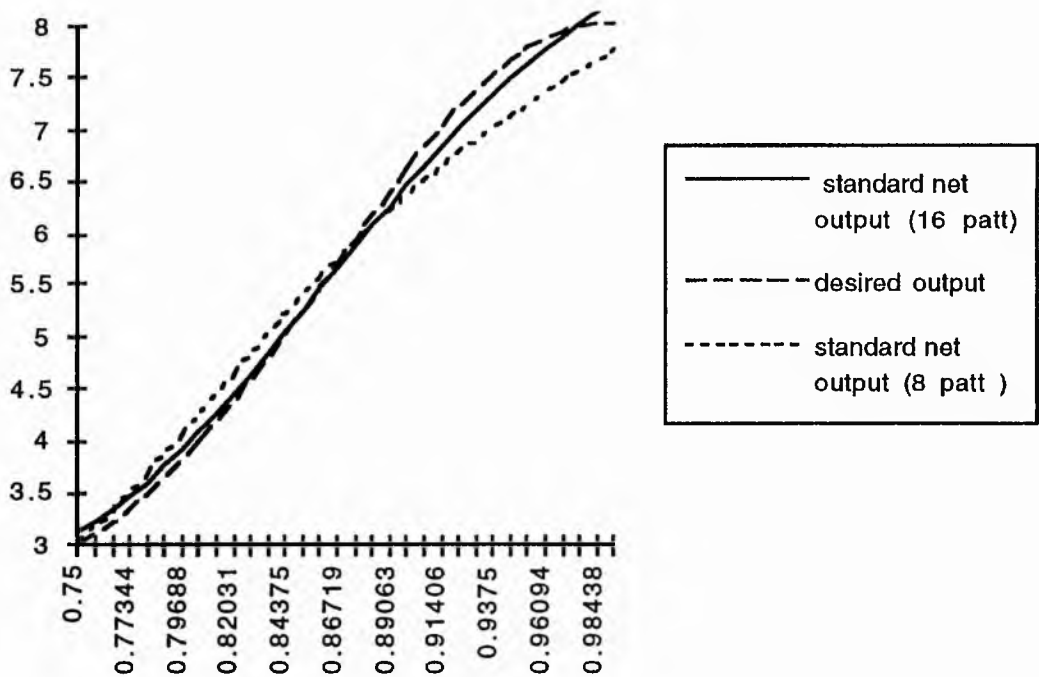


Fig. 10 The desired output function and the output of the network between 0.75 and 1. The training points in this interval are shown. Note that the functions are not equal in-between the training points.

It must be said that simple remedies like a longer training time, a smaller error limit, a larger number of points in the training set or more hidden units are not as simple as they might seem and do not necessarily work.

From the shape of the learning curve in fig. 8. it is clear that for a small decrease of the error limit, a very large increase in the training time will be necessary. However, even a much longer training time will only affect the relationship between the intersection points of the graphs in fig. 10 and the training points. Thus, for zero error, the graphs will intersect exactly in the training points. However, zero error on the training set, does not offer any guarantees with respect to the behaviour of the function in-between the data points. There is no a priori reason to believe that if the output of the network coincides with the desired output function in the training points, they will coincide in-between these training points as well.

A larger number of points in the training set will improve generalisation by ensuring that the output of the network coincides with the desired function in more points. However, increasing excessively the number of patterns in the training set can raise training problems and, unless the number of patterns tends to infinity, cannot guarantee the correspondence between the two functions.

Increasing the number of hidden units for the same number of patterns in the training set (number of samples of the desired function) can lead to overfitting phenomena. Furthermore, if both the number of hidden units and the number of patterns are increased the training can become difficult.

In conclusion, there is no simple method to get the output of the standard network closer to the desired output.

#### The complex backpropagation network

Due to the simple form of the chosen signals, particularly suited to Fourier analysis, the Fourier transform could be calculated in a straightforward manner. Then, the essential harmonics and their amplitudes could be equally easily calculated. However, in order to illustrate the method for more complicated signals and/or for signals given only by a number of samples, a different approach, more general, was followed.

The signals were sampled and a number of samples was obtained. Once more, obvious information has been deliberately ignored and the number of samples was chosen to be 16, although only 4 would have been sufficient for the given samples

(2 samples per period for the highest frequency (the number of samples calculated from the Nyquist sampling frequency)). The 16 samples of the input and output signal are presented in table 2 and are the same as those used to train the standard network.

A Fast Fourier Transform (FFT) was then applied to the samples of the functions. The same FFT would have been applied if the functions were given through a number of samples only (with an unknown analytic expression). The frequencies with both real and imaginary coefficients equal to zero were ignored and the coefficients corresponding to the others were used as the input and the output of the network. In practice, a noise limit can be used. The frequencies with both the real and the imaginary coefficients below this limit will be considered noise and ignored.

no. of samples	frequency	input		output	
		real	imaginary	real	imaginary
0	0	4	0	8	0
1	1	2	4	8	6
2	2	6	-8	4	-2
3	3	0	0	0	0
4	4	0	0	0	0
5	5	0	0	0	0
6	6	0	0	0	0
7	7	0	0	0	0
8	8	0	0	0	0

Table 5. The results of the FFT transform. The values represent the coefficients of the cos (real) and sin (imaginary) terms in (1) scaled by a factor of  $\sqrt{N}/2$  where  $N$  is the number of samples (16). The imaginary values have reverse signs. The value corresponding to the DC level is scaled by  $\sqrt{N}$ .

The results of the FFT transform are given in table 5. The values represent the coefficients of the cos (real) and sin (imaginary) terms in (1) scaled by a factor of  $\sqrt{N}/2$  where  $N$  is the number of samples ( $N=16$ ). The imaginary values have reverse signs. The value corresponding to the DC level is scaled by  $\sqrt{N}$ . A detailed

explanation of the FFT and the relationship between the FFT and the analogue Fourier transform of a function can be found in [Brigham, 1974].

The next step in the pre-processing is a scaling of the values obtained from the FFT. This is necessary because of the limited range of the output values of the complex neurons. The radial values have a range from 0.5 to 1. In order to improve the training characteristics by avoiding the flat zone on the sigmoid's graph, the value in table 2 were linearly mapped onto (0.55,0.95) instead of (0.5,1.0). The signals have non-negligible components on 3 frequencies: 0 (the constant term in expression 1),  $f_0$  and  $2f_0$ . These frequencies were mapped onto  $-\pi/2$ , 0 and  $\pi/2$  phase values. Thus, the phase channel is used to convey information about which components are present in the signal and the radius channel is used to convey information about the magnitude of these components. The linear mapping used to map the interval (a,b) onto interval (c,d) was:

$$f(x) = \frac{c-d}{a-b}x + \frac{ad-bc}{a-b}, f:[a,b] \rightarrow [c,d]$$

and the mapped values are given in table 6.

Since there are 6 input values to be associated to 6 output values, an architecture with 6 neurons on the input layer and 6 neurons on the output layer was chosen. Since the task of the network is to associate one complex input pattern to one complex output pattern, no hidden units are necessary. The values used to train the network are given in table 7.

original input values	original output values	mapped input values	mapped output values
4	8	0.85	0.95
0	0	0.75	0.75
2	8	0.8	0.95
4	6	0.85	0.9
6	4	0.9	0.85
-8	-2	0.55	0.7

Table 6. The linear mapping used to bring the desired radius values into the range of the output of the neurons.

The training converged in 4 out of 5 trials. In one trial, one of the outputs got stuck with a radial output value of 1.0 probably because of saturation. The results of the 5 trials (number of epochs and error at the end of the training) are presented in table 8.

	input		output	
unit	radius	phase	radius	phase
1	0.85	-1.5707963	0.95	-1.5707963
2	0.75	-1.5707963	0.75	-1.5707963
3	0.8	0	0.95	0
4	0.85	0	0.9	0
5	0.9	1.57079633	0.85	1.57079633
6	0.55	1.57079633	0.7	1.57079633

Table 7. The actual input and output complex values used for the training.

initial weight state	number of epochs	error at the end of training
1	141,000	<0.00000001
2	>1,474,000	>0.05
3	36,000	<0.00000001
4	40,000	<0.00000001
5	26,000	<0.00000001

Table 8. The results of 5 training trials

At the end of the training, the complex network contains in its weights all the information necessary to associate the input signal to the output signal. The pre-processing presented allows the input signal to be given either in analytical form or as samples. If the input signal is given in analytical form, it is sampled first and then a FFT is calculated. If samples are given, the FFT can be calculated directly. Note that the result of the pre-processing does not depend in principle on the number of samples (as long as this number of samples is greater than the minimum number of samples requested by the Nyquist frequency). In practice, there is a scaling factor which depends on the number of samples  $N$  but this scaling factor does not affect the shape of the signals.



Since a FFT is part of the pre-processing, all the precautions necessary for a FFT are necessary here as well. For instance, the signal is assumed to be periodic. If the signal is only given over a bounded interval, the function will be assumed to be periodic with the period equal to the length of the interval. In this case, the value of the function at the end of the interval has to be equal with the value of the function at the beginning of the interval. If this is not true, various filters can be applied. For more information about various conditions necessary for applying a FFT and how to cope when they are not satisfied see [Brigham, 1974].

The output of the CBP network is presented in table 9.

	input		output	
unit	radius	phase	radius	phase
1	0.85	-1.5707963	0.95000014	-1.5707963
2	0.75	-1.5707963	0.75000000	-1.5707963
3	0.8	0	0.95000000	0
4	0.85	0	0.90000000	0
5	0.9	1.57079633	0.85000000	1.57079633
6	0.55	1.57079633	0.70000000	1.57079633

Table 9. The input/output behaviour of the trained complex backpropagation network.

These values can be transformed through the inverse of the mapping function into coefficients of the parametric representation which can be used to plot the graph of the output function (see table 10).

net input	network output	mapped input	mapped output
0.8500000	0.95000014	4.0000000	8.0000056
0.7500000	0.75000000	0.0000000	0.0000000
0.8000000	0.95000000	2.0000000	8.0000000
0.8500000	0.90000000	4.0000000	6.0000000
0.9000000	0.85000000	6.0000000	4.0000000
0.5500000	0.70000000	-8.0000000	-2.0000000

Table 10. The inverse linear mapping used to bring the network output values back in the original range.

The output function given by the parametric representation implemented by the network is:

$$f(t) = 2.0000014 + 4 * \cos(2\pi f_0 t) - 3 \sin(2\pi f_0 t) + 1 \cos(2\pi 2 f_0 t) + \sin(2\pi 2 f_0 t)$$

The values of the coefficients are given with 7 decimal places.

The graphic representation of this function is presented in fig. 11. A zoomed-in picture of the portion between 0.75 and 1 is presented in fig. 12. This figure is on the same scale as fig. 10 on which the differences between the output of the standard network and the desired function were noticeable. The average absolute errors between values of the desired output function and the network's model for the same time values in 128 points between 0 and 1 and the respective relative errors are given in table 11.

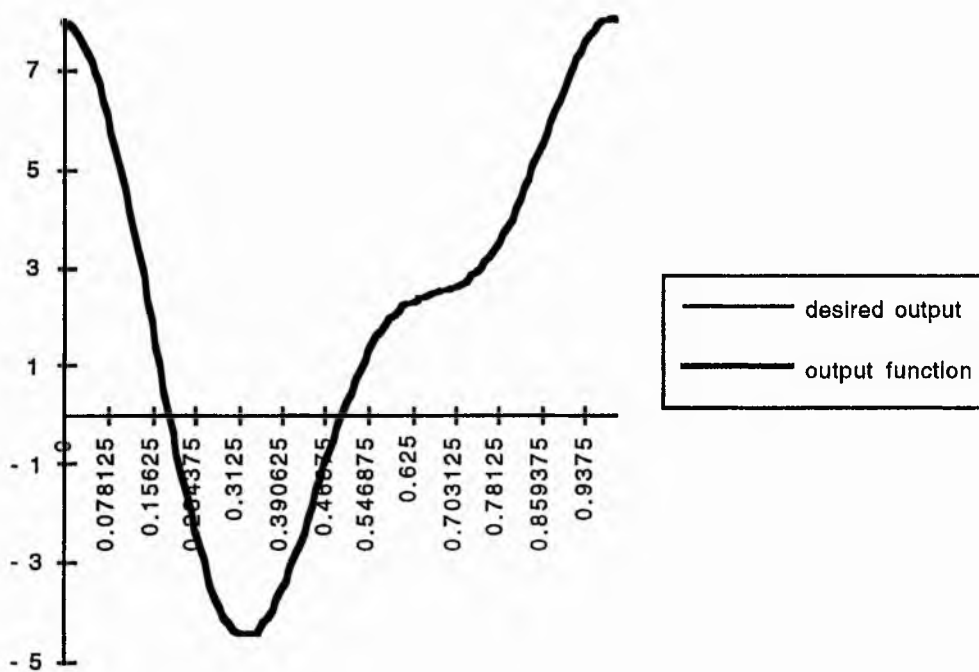


Fig. 11 The graphic representations of the desired output and the output function obtained from the output of the network are indistinguishable at this scale.

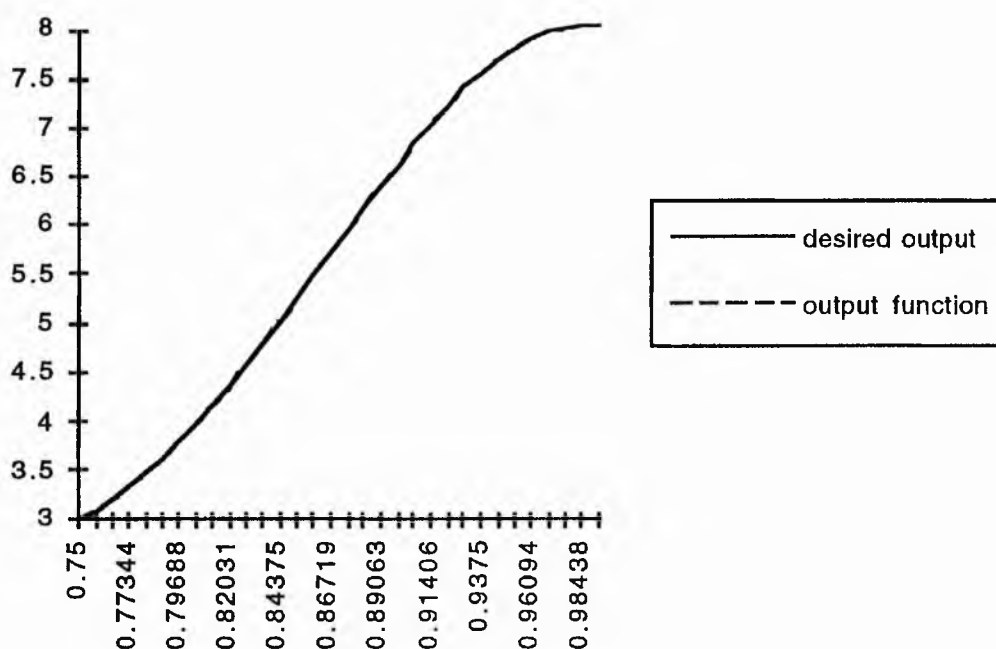


Fig. 12 The graphic representations of the desired output and of the output function between 0.75 and 1. The two functions cannot be distinguished on this scale either. For comparison see fig. 10.

	Average
average absolute error complex backpropagation	1.74E-06
average relative error complex backpropagation	0.0001%
average absolute error standard backpropagation (16 patterns)	7.46E-02
average relative error standard backpropagation (16 patterns)	3.55%
average absolute error standard backpropagation (8 patterns)	0.129985
average relative error standard backpropagation (8 patterns)	5.9819%

Table 11 Error comparison between complex backpropagation and standard backpropagation

Now, let us recall the generalisation issues discussed in chapter 3. Because we know the desired function, we can conclude that the generalisation (i.e. the behaviour of the function in-between the data points) offered by the complex network is better than the generalisation offered by the standard network. Our purpose is only to compare the two approaches not to assess them on an absolute scale. Thus, there could be many applications in which the standard approach could be more than sufficient with its 3.5% to 6% errors. In such cases, the extra pre-processing needed by the complex network and perhaps its more difficult implementation are not justified. However, there could be applications in which an improvement from 3.5 % to 0.0001% could make the difference between success and failure and in those cases, the extra effort needed by the complex network can be justified.

The fact that the complex backpropagation network has been trained to a very small error rate whereas the standard backpropagation network has been trained to a larger error rate can be seen apparently as the cause of the better generalisation exhibited by the complex network. This is not the case. As discussed, continuing training with the standard network will only improve the error on the training instances not in-between them. On the other hand, training the complex network until a small training error is achieved means that the parametric values of the network's representation are improved. In turn, this leads to a more precise overall similarity between the model of the network and the desired function.

#### Training comparison between standard and complex backpropagation

Although not essential for the purpose of the experiment which was to investigate the generalisation properties of the complex network, a brief comparison between the training properties of the two algorithms will be given in the following. The main purpose of this comparison is to show that, although the expressions of the gradient for the complex network are much more complicated than those for the standard network, the complex backpropagation training is not excessively expensive in computational terms. In other words, the price of the improvement in generalisation brought by the complex backpropagation is not an excessively heavy training. Since the two training methods are not similar at all in their computation, the comparison methods discussed in chapter 2, section 2.5 will be used in order to perform this comparison.

The expression of the gradient for the standard network is:

$$\Delta w = -\eta \frac{\partial E}{\partial w} = -\eta \delta_{out}$$

This is taken to represent 3 operations (two multiplications and the evaluation of  $\delta$ ).

The equivalent expressions for the complex network are:

$$\left\{ \begin{array}{l} \frac{\partial E}{\partial w_{jk}^{ra}} = \delta_k^{ra} r_j \cos(w_{jk}^p \theta_j - ex_k^p) + \delta_k^p \frac{r_j}{ex_k^{ra}} \sin(w_{jk}^p \theta_j - ex_k^p) \\ \frac{\partial E}{\partial w_{jk}^p} = \delta_k^{ra} r_j w_{jk}^{ra} \theta_j \sin(-w_{jk}^p \theta_j + ex_k^p) + \delta_k^p \frac{r_j}{ex_k^{ra}} w_{jk}^{ra} \theta_j \cos(w_{jk}^p \theta_j - ex_k^p) \end{array} \right.$$

$$\Delta w^{ra/p} = -\eta \frac{\partial E}{\partial w^{ra/p}}$$

which need about 21 operations. An operation is considered to be a floating point multiplication/division or a function evaluation.

A 2-12-1 standard architecture has 49 weights: 2\*12(first layer of weights)+12(second layer of weights)+13(bias). A 6-6 complex architecture has 6\*6+6=42 complex weights. The standard network used 16 real patterns (2 inputs and 1 output) whereas the complex network only used one complex pattern (6 complex inputs and 6 complex outputs). The results are summarised in table 12

	number of epochs	number of patterns	number of weights	number of operations per weight	total number of operations
standard	447,000	16	49	3	1.0513*10 <sup>9</sup>
complex	185,000	1	42	21	1.6317*10 <sup>8</sup>

Table 12 Comparison of the computational effort needed to train the standard and the complex network

The average number of epochs used by the complex backpropagation was calculated with the formula given in chapter 2, section 2.5.2 where the cost of a failure (or the number of epochs after which a training is declared unsuccessful) was taken to be 500,000 epochs (approximately three times the length of the longest

successful trial). The number of epochs necessary for each trial of the complex backpropagation are given in table 8.

From table 12, it is clear that the computational effort is of comparable orders of magnitude. Actually, for the values of the error limits used in this experiment, the training of the complex network was cheaper in computational terms than the training of the standard network but this was not considered to be either significant or typical.

### 8.5.1.2 The association of two signals

The task is to learn an amplitude independent association between two signals. Thus, two signal pairs will be trained ( $k_1 \cdot s_1$ ,  $k_1 \cdot s_2$ ) and ( $k_2 \cdot s_1$ ,  $k_2 \cdot s_2$ ). Subsequently, the net will be tested with the input signal  $k_3 \cdot s_1$ . The output of the network will be compared with the desired output for this signal which is  $k_3 \cdot s_2$ .

The first signal pair is as follows. The input signal  $k_1 s_1$  (for  $k_1=1$ ) is :

$$k_1 \cdot s_1(t) = 1 + \cos(2\pi f_0 t) - 2\sin(2\pi f_0 t) + 3\cos(2\pi 2f_0 t) + 4\sin(2\pi 2f_0 t)$$

and the output signal  $k_1 s_2$  is:

$$k_1 \cdot s_2(t) = 2 + 4\cos(2\pi f_0 t) - 3\sin(2\pi f_0 t) + 2\cos(2\pi 2f_0 t) + \sin(2\pi 2f_0 t)$$

The second signal pair is as follows. The input signal  $k_2 s_1$  (for  $k_2=0.5$ ) is :

$$k_2 \cdot s_1(t) = 0.5 + 0.5\cos(2\pi f_0 t) - \sin(2\pi f_0 t) + 1.5\cos(2\pi 2f_0 t) + 2\sin(2\pi 2f_0 t)$$

and the output signal  $k_2 s_2$  is:

$$k_2 \cdot s_2(t) = 1 + 2\cos(2\pi f_0 t) - 1.5\sin(2\pi f_0 t) + \cos(2\pi 2f_0 t) + 0.5\sin(2\pi 2f_0 t)$$

After training, both the complex network and the standard one will be tested with the input signal  $k_3 s_1$  (for  $k_3=0.75$ ):

$$k_3 \cdot s_1(t) = 0.75 + 0.75\cos(2\pi f_0 t) - 1.5\sin(2\pi f_0 t) + 2.25\cos(2\pi 2f_0 t) + 3\sin(2\pi 2f_0 t)$$

The desired output signal  $k_3 s_2$  (for the same  $k_3=0.75$ ) is:

$$k_3 \cdot s_2(t) = 1.5 + 3\cos(2\pi f_0 t) - 2.25\sin(2\pi f_0 t) + 1.5\cos(2\pi 2f_0 t) + 0.75\sin(2\pi 2f_0 t)$$

If the networks yield the desired output signal for the untrained input signal, this will support the belief that the networks implement the desired amplitude independent association between the signals  $s_1$  and  $s_2$ .

The I/O patterns were obtained for both the complex and the standard network by pre-processing the signals in the corresponding manners described in the previous section.

The standard backpropagation was trained in three different sessions to a maximum absolute error of 0.104825, 0.089155 and 0.037718 respectively. The complex backpropagation was trained to an maximum absolute error limit of 0.026. Table 13 presents the average (over 128 test points) absolute error and the average relative error (with respect to the amplitude of the desired function) of the function yielded by the complex network and the standard network.

A graphical representation of the functions obtained through using the complex and standard networks is given in fig. 13. Fig. 14 presents the same functions in some more detail, on a sub-interval (0.75, 1) of the definition interval (0, 1).

It is interesting to note that for the standard backpropagation, improving the error on the training set, actually made the generalisation worse. Thus, although the error at the end of the last training session is the lowest, the generalisation given by the correspondent weight state is the worst. This can be seen as a typical example of overtraining in which the seek for a low error limit on the training set made the network to implement an I/O function which oscillates very badly in-between the training points.

The complex backpropagation network does not behave this way because a smaller error in the complex training corresponds to a closer approximation of the parameters of the desired I/O function as opposed to a closer approximation of the function itself in the training points only. Such a better approximation of the parameters corresponds to a better overall approximation of the function.



	average
absolute error complex backpropagation	0.05885001
absolute error standard backpropagation ( 1 )	0.47001433
absolute error standard backpropagation ( 2 )	0.55385955
absolute error standard backpropagation ( 3 )	1.0447698
relative error complex backpropagation	1.729583%
relative error standard backpropagation ( 1 )	8.360436%
relative error standard backpropagation ( 2 )	7.448624%
relative error standard backpropagation ( 3 )	20.283869%

Table 13. The average absolute and relative errors of complex and standard training sessions. These data have been obtained by comparing the output yielded by the network with the desired output when the networks were fed with the untrained test input.

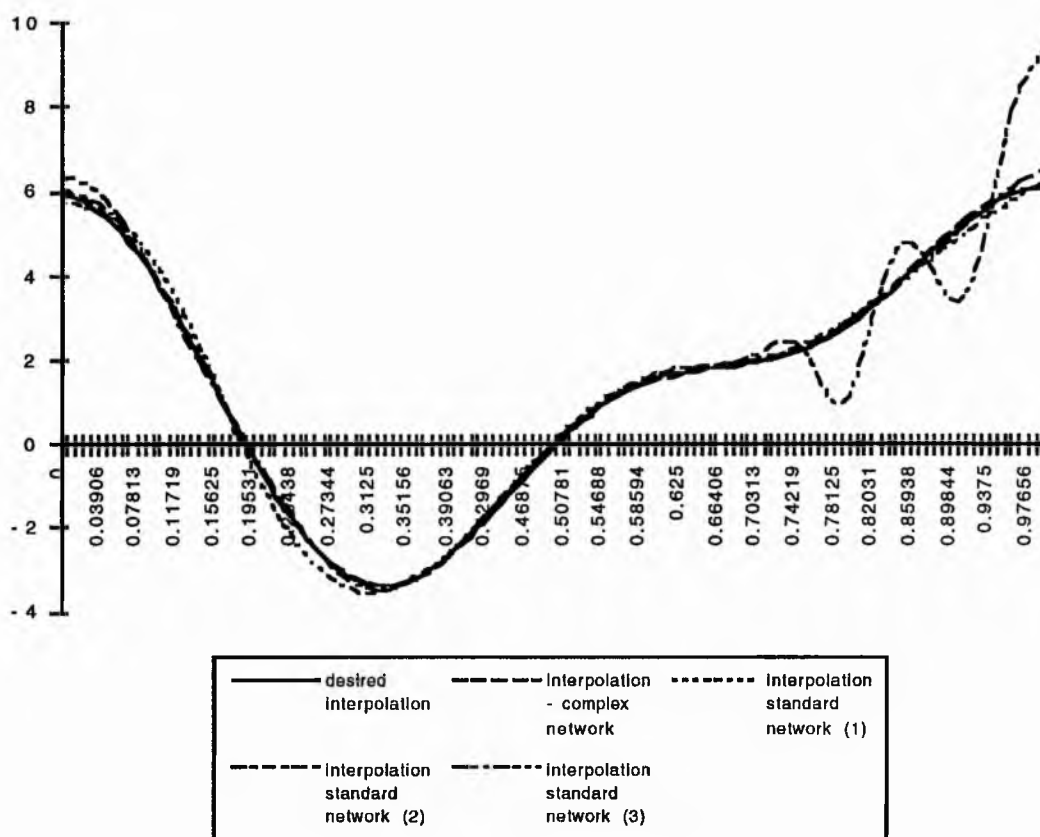


Fig. 13. The desired function compared with the functions yielded by the complex and standard networks on the entire definition interval.

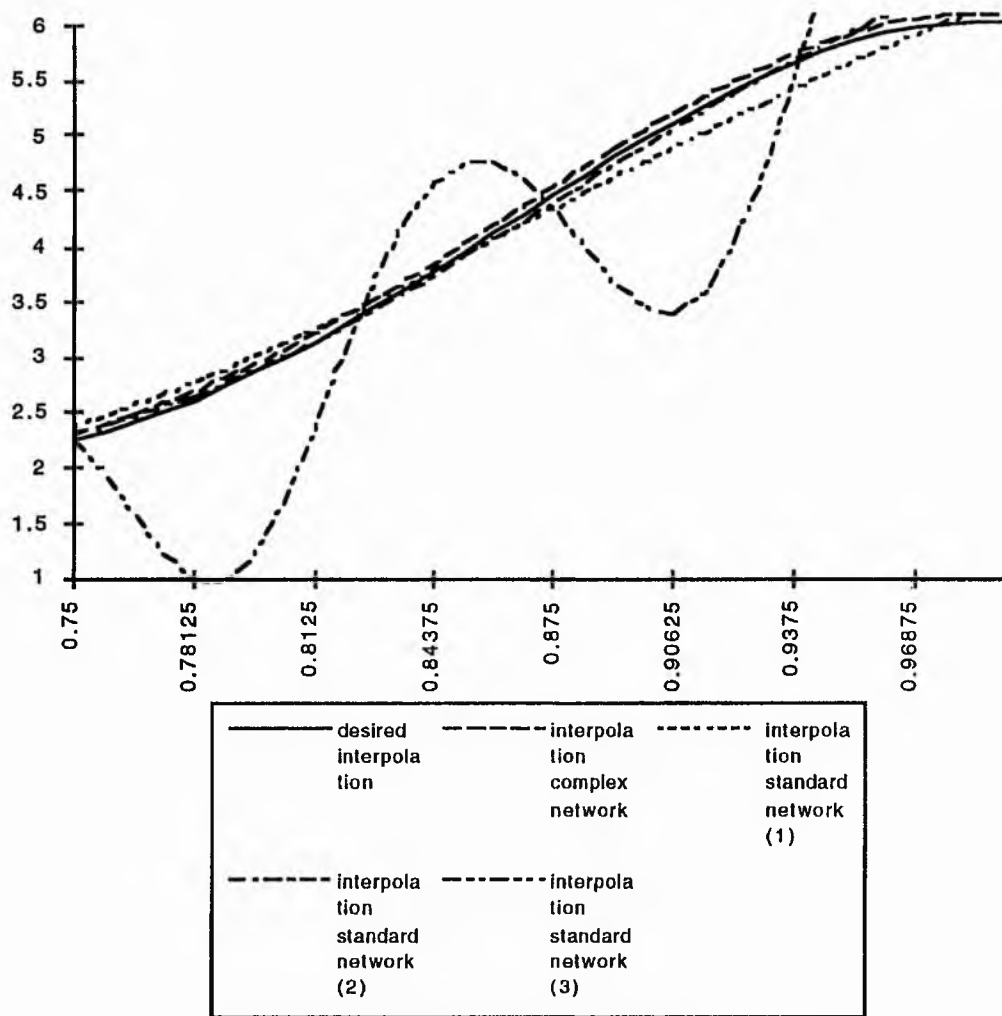


Fig. 14. The desired function compared with the functions yielded by the complex and standard networks on a sub-interval of the definition interval.

## **8.6. Conclusions**

### **8.6.1. Testing the training algorithm.**

If the architecture is very simple (one link without bias or one link with bias) and there are sufficient patterns in the training set, the training algorithm manages to rediscover the original weight state (in 3 out of 5 trials) or fails altogether (2 out of 5). These results were interpreted as indicating a complicated error surface which caused the failures but a correct training algorithm which allowed the network to retrieve the original weight states when the initial weight state was not particularly bad.

If the architecture is rich, the training algorithm manages to find a solution which gives a reasonably small error. However, the difficulty of the training increases with the complexity of the architecture. This behaviour is similar to the behaviour of the standard training and it is usually regarded as being a consequence of too large a number of parameters (or degrees of freedom).

For a slightly more complicated architecture such as a 1-1-1 with bias (two layers of trainable weights with one neuron on the hidden layer) or 1-2-1 with bias (two layers of trainable weights with two neurons on the hidden layer) the training algorithm does not find the original solution but it is still able to find a low (or very low such as 0.00000001) error solution.

### **8.6.2. Testing the training algorithm with a random I/O problem.**

Training 4 random patterns with a 1-2-1 and 1-3-1 architecture was successful as far as finding a low-error weight state is concerned. However, the training times for these random I/O problems are long.

In conclusion, the training algorithm seems to behave in a predictable manner, rather similar to that of the standard backpropagation.

### **8.6.3. Testing the general approach**

A simple example was used to illustrate the method for associating signals using a network with complex weights. The experiments showed that it is possible to use such a network to store a parametric representation of a signal.

This approach is able to exploit a good precision at the level of the network (lower error limits than those currently used with standard backpropagation). This is because the numerical values resulted from the network are interpreted as parametric values of signals and not as instantaneous values. In the case of the experiments presented, such a good precision could be obtained with a computational effort comparable to that of the standard network trained with standard backpropagation.

The experiments showed that the parametric representation approach can offer a better generalisation than the standard approach of training I/O samples. The improvement is given by the global approach to associating signals and not by the lower error limits used.

Another positive characteristic of the parametric approach is that it can be used with very few patterns and units if the appropriate parametric representation is chosen and this is independent of the degree of the non-linearity of the function to be implemented. This is different from the behaviour of the standard approach which needs more samples as the degree of the non-linearity of the function increases and exhibits a more difficult training when more samples are present.

However, the parametric approach has weaknesses as well. Thus, elements such as the specific parametric representation or appropriate pre-processing are essential. If the chosen parametric representation is not able to model well the class of phenomena it is used for, the generalisation will be poor. For instance, if the complex network is used to model a constant function and the pre-processing used is that presented above (based on an FFT), the results will be worse than those obtained with the standard backpropagation. This is because, the Fourier representation of a constant signal has an infinite number of terms thus requiring an infinite number of input and output complex units for a zero error representation. At the same time, a standard backpropagation network has no trouble in representing perfectly a constant signal. Indeed, such a signal can be easily represented by a network with a single non-zero weight (with an appropriate value) from the bias neuron to the output neuron.

If no information on the class of phenomena to be modelled is given, the choice of the appropriate parametric representation is not trivial. However, general rules of thumb can be imagined. Thus, for periodic, infinite signals, a Fourier parametric representation is very well suited. This is because the basis functions used in this parametric representation (sin and cos functions) are infinite in the time domain and

localised in the frequency domain. This allow the parametric representation of infinite signals to be very compact and very efficient.

For functions which are localised in time domain an alternative parametric representation can be obtained through a wavelet transform for instance ([Press, 1992], [Daubechies,1992]). The wavelet basis functions can be more appropriate in this case because they have the property of being localised both in time and frequency domains.

However, the complex network presented can be used with a variety of parametric representations which can be chosen to fit a particular application.

# CHAPTER 9

## Conclusions and further work

### 9.1 General remarks

The objective of the research work described in this thesis was to understand and further investigate certain aspects of artificial neural networks. Issues like training and generalisation were reviewed and discussed.

The discussion of the training issues set a goal for a good training algorithm: to combine the convergence properties of the perceptron training with the capabilities of a more complicated architecture. Furthermore, issues regarding the assessment of training speed were discussed and a new way of assessing the training speed was proposed. This new measure allows the comparison of non-similar training algorithms in a machine-independent fashion.

Various definitions of generalisation were analysed and some difficulties regarding generalisation assessment were emphasised. Although no fundamentally new facts were presented, the conclusions of this discussion might appear surprising or at least non-trivial to some neural network researchers. Thus, one of the conclusions of this chapter is that no network architecture or training algorithm can be said to offer a "good generalisation" or a "better generalisation" unless the problem is clearly stated and there is class of known desired underlying functions. However, there are many cases in the literature when techniques are claimed to offer "good generalisation" without specifying the particular assumptions made.

Constraint Based Decomposition was presented as a new divide-and-conquer approach to training neural networks. It has been shown that it is useful to regard any training algorithm as a pair formed of a weight changing mechanism and a pattern presentation algorithm. It has also been shown that the pattern presentation algorithm, mostly neglected in the literature, is important for the outcome and speed of the training. The CBD approach used only at the level of the pattern presentation algorithm was shown to be able to improve the training of a multilayer perceptron. However, the improvement brought by the CBD pattern presentation algorithm was assessed as not sufficiently reliable.

A new constructive algorithm based on the same CBD approach was then presented. This algorithm was shown to be able to solve difficult problems like the

2-spirals whilst having a good training speed and guaranteed convergence. However, the standard version of the CBD constructive algorithm can use redundant hyperplanes and can waste time in subgoal training sessions which cannot be solved. These deficiencies are corrected by the enhanced version of the algorithm.

The CBD constructive algorithm was investigated from the point of view of generalisation as well. From this point of view, the algorithm was shown to have interesting properties: the shape of the solution can be influenced in a controlled manner by the order of the patterns in the training set. It has been shown that optimal solutions (using the minimum number of hyperplanes) with desired characteristics can be consistently obtained by controlling the order of the patterns in the training set.

The performances of the CBD constructive algorithm compared favourably with both the performances of the multilayer perceptron trained with standard backpropagation with momentum and other constructive techniques like divide and conquer networks.

The generalisation issues were addressed by proposing the idea of feeding the network not only with local information represented through some sample point values but with some a priori knowledge about the underlying function as well in those cases in which such information exists. One possibility for doing this was presented, namely a signal space approach, in which the network works in a parametric space of functions. The description of this parametric space is done such that it conveys the available a priori information about the underlying function through the choice of its basis. The network is used to associate the parameters of the input function with the parameters of the output function. Due to this feature, the generalisation offered by the solution weight state is better than the one offered by the classical approach of training samples. Diminishing the error in the parameter training means a better overall correspondence between the underlying function and the function implemented by the network. This is unlike the standard approach of training samples where diminishing the error means a better correspondence between the underlying function and the function implemented by the network exclusively on the training points.

A neural network was designed for the above purpose. This network has two channels, one for the basis functions of the signal space and the other one for the co-ordinates along the respective axes. The cross-talking between channels was not



considered detrimental to the purpose of the network and was therefore allowed. A network with complex weights in radial and phase form was used to implement and test the concept.

Some simple experiments were used to test the reliability of the complex network as designed and the validity of the approach. The approach was shown to yield better generalisation than the standard approach in the experiments performed.

## **9.2. Limitations**

As with all techniques, the techniques presented in this thesis have limitations. The main limitations and further work needed to remove them and further explore the techniques are presented in the following.

### **9.2.1 Constraint Based Decomposition**

CBD at the level of the pattern presentation algorithm only is not a reliable technique as yet. Its improvement over the batch or sequential (on-line updating) pattern presentation algorithms is not guaranteed and can be negligible for some problems.

Furthermore, the subgoal definition is not straightforward. At the moment, this subgoal definition depends on the problem and has to be done manually.

The constructive CBD algorithm has been studied only in classification problems. Further work is necessary to explore its performances in problems with analogue targets.

There is no known automatic or simple and general method to find a good ordering of the patterns in the training set given the patterns and the desired I/O function. At the moment, a good ordering has to be found manually, for each problem. This process is likely to be difficult for some problems.

### **9.2.2 The Complex Backpropagation**

The approach is expected to be better than the standard approach only if there is some a priori knowledge about the underlying function or if a certain type of generalisation is preferred. An appropriate set of basis functions in signal space has to be chosen and this set has to be able to model well the desired type of functions. A basis formed with sin and cos functions (i.e. a Fourier signal space) can be used for a large category of functions as shown by the examples presented. Another possibility can be a set of wavelet basis functions.

The approach involves a certain amount of pre-processing.

### **9.3 Further work**

There are some topics for further investigation which appear natural and/or particularly inviting. The results of further research along these lines could eliminate some of the limitations discussed above or could further develop the work presented in this thesis. A few such topics of further research will be presented in the following.

#### **9.3.1 The Constraint Based Decomposition**

CBD at the level of the pattern presentation algorithm would benefit from an automatic method to divide the pattern set into subgoals for use in conjunction with standard weight changing mechanisms.

The constructive CBD algorithm could be augmented by investigating its behaviour in problems with analogue targets.

An algorithm for symbolic reduction of logical expressions as another form of redundancy elimination could be designed perhaps. Such an algorithm would eliminate redundant terms in the AND and OR layers and help obtaining a more compact solution in certain cases.

The linear separability detection mechanism needs implementation and testing for the  $n$ -dimensional case.

An algorithm for automatic ordering could be perhaps designed. Such an algorithm would take into consideration a priori information about the desired I/O mapping and use it to find a pattern ordering which would determine the CBD constructive algorithm to construct a network yielding the desired generalisation.

#### **9.3.2 Complex backpropagation**

Further investigation of the reduced training speed of the complex network and possible improvements can be done. Once a training regime with better characteristics is available, applying the technique to real-world problems becomes more feasible.

Other possibilities to convey magnitude/component information and other options for the complex neurons can be imagined. One of these possibilities is to use

networks using neurons formed by pairing real-valued neurons. Different types of processing at the level of the neuron could also be imagined.

An investigation of other types of signal space is very appealing. Different, corresponding types of pre-processing (such as wavelet analysis) are required and could bring substantial improvements.

## Bibliography

- [Ackley, 1985] - Ackley D. H., Hinton, G. E., Sejnowski T. J. - A learning algorithm for Boltzmann machines, *Cognitive Science* 9, pp. 147-169, 1985
- [Alstrom, 1994] - Alstrom P., Stassinopoulos D. - Adaptive Performance Networks, submitted to *Phys. Rev. Lett.*, 1994.
- [Amari, 1993] - S. Amari, A universal theorem on learning curves, *Neural Networks*, vol. 6, pp. 161-166, 1993
- [Anderson, 1972] - Anderson J.A. - A simple neural network generating an interactive memory, *Mathematical Biosciences* 14, pp. 197-220.
- [Anderson, 1988] - Edited by Anderson J.A. and Rosenfeld E. - *Neurocomputing: foundations of research*, MIT Press, 1988.
- [Andes, 1990] - D. Andes, B.Widrow, M.Lehr, E.Wan, MRIII: A robust algorithm for training analog neural networks, *IJCNN*, vol.I, pp. 553-536, 1990
- [Atkin, 1989] - Atkin, G.K., J.E. Bowcock and N.M. Queen - Solution of a distributed deterministic parallel network using simulated annealing. *Pattern Recognition* 22 461-466 (1989).
- [Baba, 1989] - N. Baba, A New Approach for Finding the Global Minimum of Error Function of Neural Networks, *Neural Networks*, vol. 2, pp. 367-373, 1989
- [Baffes, 1992] - Baffes, P.T., J.M. Zelle - Growing layers of perceptrons: introducing the extentron algorithm
- [Baum, 1989] - E. Baum, D. Hausller - What size net gives valid generalisation?, *Neural Computation* 1, pp. 151-160, 1989
- [Baum, 1990] - Baum, E. B., K. J. Lang - Constructing hidden units using examples and queries, *NIPS 1990*, pp. 904-910.
- [Baum, 1991] - Baum, E. B. - Neural net algorithms that learn in polynomial time from examples and queries, *IEEE Transactions of Neural Networks* 2(1), January, 1991.

[Bellman, 1957] - R. Bellman, Dynamic Programming, Princeton University Press, Princeton, N.J., 1957

[Benvenuto, 1991] - N. Benvenuto, M. Marchesi, F. Piazza, A. Uncini, A comparison between real and complex valued neural networks in communication applications, Artificial Neural Networks, T. Kohonen, K. Makisara, J. Kangas (Editors), Elsevier Science Publishers, 1991

[Benvenuto, 1992] - N. Benvenuto, F. Piazza, On the complex backpropagation algorithm, IEEE Transactions on Signal Processing, vol. 40, no. 4, April 1992

[Birx, 1989] - D.L. Birx, S.J. Pipenberg, Neural Network Structures for Defect Discrimination, presented at the Fifth Annual Aerospace Applications of Artificial Intelligence Conference, Dayton, OH, 24-26 Oct. 1989

[Bishop, 1993] - C.M Bishop, Neural Network Validation: an Illustration from the Monitoring of Multi-phase Flows

[Block, 1962] - H.D. Block, The perceptron: a model for brain functioning, Reviews of Modern Physics 34, 123-135

[Brady, 1988] - M. Brady, R. Raghavan, Gradient descent fails to separate, Proceedings of the IEEE International Conference on Neural Networks, vol. 1, pp. 649-656, 1988

[Breiman, 1984] - Breiman L., J. H. Friedman, R. A. Olsen, C. J. Stone - Classification and regression trees, Wadsworth & Brooks, (1984).

[Breiman, 1993] - L. Breiman, J.H. Friedman, R.A. Olshen, C.J. Stone - Classification and Regression Trees, Chapman & Hall, 1993

[Brent, 1991] - R. P. Brent, Fast Training Algorithms for Multilayer Neural Nets, IEEE Transactions on Neural Networks, vol. 2, no. 3, May 1991

[Brigham, 1974] - O. Brigham, The Fast Fourier Transform, Prentice Hall, 1974

[Broomhead, 1988] - D. S. Broomhead, D. Lowe, Multivariable functional interpolation and adaptive networks, Complex Systems 2, 321-323, 1988

[Brunak, 1990] - Brunak S., Lautrup B. - Neural networks: Computers with intuition, Singapore: World Scientific, 1990

[Bryson, 1969] - Bryson A.E., Ho Y.-C. - Applied Optimal Control, New York: Blaisdell, 1969.

[Burrascano, 1990a] - P. Burrascano, P.Lucci, A learning rule in the Chebyshev norm for multilayer perceptron, IJCNN, vol.III, pp. 81-86, 1990

[Burrascano, 1990b] - P. Burrascano, P.Lucci, Smoothing backpropagation cost function by delta constraining, IJCNN, vol.III, pp. 75-80, 1990

[Chang, 1990] - E.I.Chang, R.P. Lipmann, D.W.Tong, Using genetic algorithms to select and create features for pattern classification, IJCNN, vol.III, pp.747-752, 1990

[Chen, 1992] - LiHui Chen, Modelling continuous sequential behaviour to enhance training and generalisation in neural networks, PhD thesis, University of St Andrews, 1992

[Cheng, 1989] -Cheng D.K. - Field and Wave Electromagnetics, Addison Wesley, 1989

[Clarke, 1990] - T.L. Clarke, Generalisation of Neural Networks to the Complex Plane, IJCNN 1990, pp. II 435-440

[Cotrell, 1987] - Cotrell G. W., Munro P., Zipser D. - Learning Internal Representations from Grey-Scale Images: An example of Extensional Programming, Ninth Annual Conference of the Cognitive Science Society, Seattle, 1987, pp. 462-473, Hillsdale: Erlbaum, 1987

[Courrieu, 1994] - P. Courrieu, Three algorithms for estimating the domain of validity of feedforward neural networks, Neural Networks, vol. 7, no. 1, pp. 169-174, 1994

[Daubechies, 1992] - I. Daubechies, Wavelets, SIAM 1992

[de Garis, 1990] - H. de Garis, Genetic programming - Building nanobrainns with genetically programmed neural network modules, IJCNN, vol.III, pp. 511-516, 1990

[Dembo, 1990] - A. Dembo, T.Kailath, Model-free distributed learning, IEEE Transactions on Neural Networks, vol. 1, no.1, 1990

[Demidovich, 1981] - B. P. Demidovich, I. A. Maron - Computational Mathematics, Third Edition, Mir Publishers, Moscow, 1981

[Denker, 1987] - J. Denker, D. Schwartz, B. Wittner, S. Solla, R. Howard, L. Jackel, J. Hopfield - Large Automatic Learning, Rule Extraction and Generalisation, Complex Systems 1, pp. 877-922, 1987

[Denoeux, 1993] - Denoeux T., R. Lengelle - Initialising backpropagation networks with prototypes, Neural Networks, vol. 6, 351-363, (1993).

[Dodd, 1990] - N. Dodd, Optimisation of network techniques using genetic techniques, IJCNN, vol. III, pp. 965-970

[Draghici, 1995] - Draghici S., M. K. Weir - Enhancements of the Constraint Based Decomposition, Technical Report CS/95/5, Department of Computer Science, St Andrews University, 1995

[Falhman, 1988] - S.E. Falhman, An empirical study of Learning Speed in Back-Propagation Networks, Technical report, Carnegie Mellon University, CMU-CS-88-162, (1988).

[Falhman, 1990] - Falhman, S.E., C. Lebiere - The Cascade-Correlation Learning Architecture, Technical Report CMU-CS-90-100, Carnegie Mellon University, 1990

[Farhat, 1985] - Farhat N.H., Psaltis D., Prata A., Pack E. - Optical implementation of the Hopfield model, Applied Optics 24, pp. 1469-1475, 1985.

[Fernandez, 1994] - Antonio Fernandez, Talk to the Neural Group, St Andrews University

[Frean, 1990] - Marcus Frean, The Upstart algorithm: A Method for Constructing and Training Feedforward Neural Networks, Neural Computation 2, 198-209, 1990

[Frean, 1990] - The Upstart algorithm: A Method for Constructing and Training Feedforward Neural Networks, Marcus Frean, Neural Computation 2, 198-209, 1990

[Fukushima, 1975] - Fukushima K. - Cognitron: A self-organising multilayered neural network, Biological Cybernetics 20, pp. 121-136, 1975.

[Fukushima, 1980] - Fukushima K. - Neocognitron: A self-organising neural network model for a mechanism of pattern recognition unaffected by shift in position, *Biological Cybernetics* 36, pp. 193-202, 1980

[Fukushima, 1982] - Fukushima K. - Neocognitron: A new algorithm for pattern recognition tolerant of deformations and shifts in position, *Pattern Recognition* 15, pp. 455-469, 1982

[Gallant, 1986] - Gallant S. I. - Optimal Linear Discriminants. In *Eighth International Conference on Pattern Recognition, Paris 1986*, 849-852, New York: IEEE

[Girosi, 1990] - T. Poggio, F. Girosi - Networks for Approximation and Learning, *Proceedings of IEEE*, vol. 78, no. 9, Sept. 1990

[Golea, 1990] -M. Golea, M. Marchand, A growth algorithm for neural network decision trees, *Europhysics Letters*, pp. 205-210, 1990

[Gorman, 1988] - Gorman R. P., Sejnowski T. J. - Learned Classification of Sonar Targets Using a Massively-Parallel Network, *IEEE Transactions on Acoustics, Speech and Signal Processing* 36, pp. 1135-1140, 1988

[Grossberg, 1967] - Grossberg S. - Nonlinear Difference-Differential Equations in Prediction and Learning Theory. *Proceedings of the National Academy of Sciences, USA* 58, 1329-1334.

[Grossberg, 1968a] - Grossberg S. - Some Nonlinear Networks Capable of Learning a Spatial Pattern of Arbitrary Complexity. *Proceedings of the National Academy of Sciences, USA* 59, 368-372

[Grossberg, 1968b] - Grossberg S. - Some Physiological and Biochemical Consequences of Psychological Postulates. *Proceedings of the National Academy of Sciences, USA* 60, 758-765.

[Grossberg, 1969] - Grossberg S. - Embedding Fields: A Theory of Learning with Physiological Implications. *Journal of Mathematical Psychology* 6, 209 - 239.

[Grossberg, 1972] - Grossberg S. - Neural Expectation: Cerebellar and Retinal Analogs of Cells Fired by Learnable or Unlearned Pattern Classes. *Kybernetik* 10, 49-57



[Grossberg, 1976a] - Grossberg S. - Adaptive Pattern Classification and Universal Recording: I. Parallel Development and Coding of Neural Feature Detectors. *Biological Cybernetics* 23, 121-134.

[Grossberg, 1976b] - Grossberg S. - Adaptive Pattern Classification and Universal Recording. II. Feedback, Expectation, Olfaction, Illusions. *Biological Cybernetics* 23, 187-202.

[Grossberg, 1980] - Grossberg S. - How Does the Brain Build a Cognitive Code?, *Psychological Review* 87, 1-51

[Hertz, 1991] - Hertz, J., A. Krogh, R.G. Palmer - Introduction to the theory of neural computation, Addison Wesley, 1991

[Hinton, 1981] - Hinton, G., J. Anderson - Parallel models of Associative Memory, Hillsdale, NJ: Lawrence Erlbaum Associates, (1981).

[Hirose, 1991] - Y. Hirose, K. Yamashita, S. Hijiya, Backpropagation algorithm which varies the number of hidden units, *Neural Networks*, no. 4, pp. 61-66, 1991

[Hopfield, 1982] - Hopfield J.J - Neural networks and physical systems with emergent collective computational abilities, *Proceedings of the National Academy of Sciences* 79, 2554-2558, 1982

[Hopfield, 1984] - Hopfield J.J - Neurons with graded response have collective computational properties like those of two-state neurons, *Proceedings of the National Academy of Sciences* 81, pp. 3088-3092, 1984.

[Hornik, 1989] - K. Hornik, M. Stinchcombe, H. White, Multilayer feedforward networks are universal approximators, *Neural Networks*, vol. 2, pp. 359-366, 1989

[Hornik, 1993] - K. Hornik, Some new results on neural network approximation, *Neural Networks*, vol. 6, pp. 1069-1072, 1993

[Hugill, 1985] - M. Hugill, *Advanced Statistics*, Bell & Hyman, 1985

[Hwang, 1990] - J. Hwang, J.J. Choi, S. Oh, R.J. Marks, Query learning based on boundary search and gradient computation of trained multilayer perceptrons, *IJCNN*, pp. 57-62, 1990

- [Hyafil, 1976] - L. Hyafil, R. Rivest, Information Processing Letters 5 (1976) 15
- [Jacobs, 1989] - R. A. Jacobs, Increased rate of convergence through learning rate adaptation, Neural Networks 1, 295-307, 1989
- [James, 1890] - Psychology (Briefer Course), New York: Holt, 1890.
- [Joerding, 1991] - Joerding, H.W., J.L. Meador, Encoding a priori information in feedforward networks. Neural Networks vol. 4, 847-857, (1991).
- [Jupp, 1994] - P. Jupp, Private Communication, 1994
- [Knerr, 1991] - S. Knerr, L. Personnaz, G. Dreyfus, A new approach to the design of neural network classifiers and its applications to the automatic recognition of hand-written digits, IJCNN, vol. 1, pp.91-96, 1991
- [Kohonen, 1972] - Kohonen T. - Correlation matrix memories, IEEE Transactions on Computers C-21, pp. 353-359.
- [Kohonen, 1982] - Kohonen T. - Self-organized formation of topologically correct feature maps, Biological Cybernetics 43, pp. 59-69, 1982.
- [Kramer, 1989] - A. H. Kramer, A. Sangiovanni-Vincentelli, Efficient Parallel Learnini Algorithms for Neural Networks, In Advances in Neural Information Processing Systems I, Denver 1988
- [Lang ,1988] - Lang K. J, M. J. Witbrock - Learning to tell two spirals apart, Proceedings of the 1988 Connectionist Models Summer School, Morgan Kaufmann.
- [Lapedes, 1987] - Lapedes A., Farber R. - Nonlinear Signal Processing Using Neural Networks: Prediction and System Modelling. Technical Report LA-UR-87-2662, Los Alamos National Laboratory, Los Alamos, NM, 1987
- [Le Cun, 1985] - Le Cun Y. - Une procedure d'apprentissage pour reseau a seuil assymetrique (A learning procedure for asymmetric threshold network), Proceeding of Cognitiva 85, pp. 599-604, Paris, 1985.
- [Leung, 1991] - H. Leung, S.Haykin, The Complex Backpropagation Algorithm, IEEE Transactions on signal processing, vol. 39, no. 9, September 1991

[Little, 1975] - Little W. A., Shaw G. L. - A statistical theory of short and long term memory, *Behavioral Biology* 14, pp.115-133, 1975

[Little, 1990] - G. Little, S.C. Gustafson, R.A. Senn, Generalisation of the backpropagation neural network learning algorithm to permit complex weights, *Applied Optics*, vol. 29, no. 11, April 1990

[Marchand, 1990] - M. Marchand, M. Golea, P.Rujan, A convergence theorem for sequential learning in two-layer perceptrons, *Europhysics Letters*, 11 (6), pp. 487-492, 1990

[McClelland, 1986] - McClelland, J.J., D. Rumelhart et. al. - Parallel Distributed processing, MIT Press, 1986

[McCulloch, 1943] - McCulloch, W.W, Pitts, W, A logical calculus of the idea immanent in neuron activity, *Bulletin of Mathematical Biophysics*, 1943.

[Mezard, 1989] - Mezard M., J.-P. Nadal - Learning in Feedforward Layered Networks: The Tiling Algorithm, *Journal of Physics A* 22, 2191-2204

[Minsky, 1969] - M.L. Minsky, S.A. Papert, *Perceptrons*. Cambridge: MIT Press, 1969

[Mitchell, 1977] - T. M. Mitchell, Version Spaces: A Candidate Elimination Approach to Rule Learning, *IJCAI 5*, MIT, USA, vol.1, pp.1139-1151, 1977

[Mitchell, 1977] - T.M. Mitchell, Version Spaces: An Approach to Concept Learning, PhD Thesis, STAN-CS-78-711, Stanford University, USA, 1978

[Mitchell, 1978] - Version space: An approach to concept learning, Tom M. Mitchell, PhD Thesis, Stanford University, 1978

[Mitchell, 1980] - T. M. Mitchell, The Need for Biases in Learning Generalisations, Rutgers Computer Science Tech. Report, 1980

[Mitchell, 1982] - T. M. Mitchell, Generalisation as Search, *Artificial Intelligence* vol. 18, 1982, pp.. 203-226

[Moeller, 1993] - M. F. Moeller, A scaled conjugate gradient algorithm for fast supervised learning, *Neural Networks*, vol. 6, pp. 525-533, 1993

[Moody, 1989] - Moody, J., C.J. Darken - Fast learning in networks of locally-tuned processing units. *Neural Computation* 1, 281-294, (1989).

[Murray, 1992] - A.F. Murray, Analog VLSI and multilayer perceptrons - accuracy, noise, on-chip learning, *Neurocomputing*, vol. 4, no. 6, 301-310, (1992).

[Musavi, 1992] - M.T. Musavi, W. Ahmed, K.H.Chan, K.B. Faris and D.M. Hummels - On the training of radial basis function classifiers, *Neural Networks*, vol. 5, 595-603, (1992).

[Nguyen,1990] - D.Nguyen, B. Widrow - Improving the Learning Speed of 2-Layer Neural Networks by Choosing Initial Values of the Adaptive Weights, *ICNN 1990*

[Nitta, 1993] - T. Nitta, A Back-propagation Algorithm for Complex Numbered Neural Networks, *Proceedings of 1993 IJCNN*, pp. 1649-1652

[Parker, 1985] - Parker D.B. - Learning Logic (TR-47), Cambridge, MA, MIT, Center for Computational Research in Economics and Management Science, 1985.

[Parker, 1987] - Parker, D.B. - Optimal algorithms for adaptive networks: second order backpropagation, second order direct propagation and second order Hebbian learning. *IEEE First International Conference on Neural Networks* vol. 2, 593-600 (1987).

[Plaut, 1986] - Plaut, D., S. Nowlan and G. Hinton - Experiments on learning by backpropagation. Technical Report CMU-CS-86-126, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA (1986).

[Poggio, 1990] - Poggio, T., F. Girosi - Networks for approximation and learning. *Proceedings of the IEEE*, vol. 78, no. 9, 1481-1497, September 1990.

[Poggio,1990] - T. Poggio, F. Girosi - Regularization Algorithms for Learning That Are Equivalent to Multilayer Networks, 1990

[Pomerleau, 1989] - Pomerleau D. A. - ALVINN: An Autonomous Land Vehicle in a Neural Network. In *Advances in Neural Information Processing Systems I*, Denver, 1988, ed. D.S. Touretzky, pp. 305-313, San Mateo: Morgan Kaufmann

[Preparata, 1985] - Franco P. Preparata, Michael I. Shamos, Computational Geometry - An Introduction, Springer-Verlag, 1985

[Press, 1992] - W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery, Numerical Recipes in C - The Art of Scientific Computing, second edition, Cambridge University Press, 1992

[Ravindran, 1987] - A. Ravindran, D.T. Philips, J.J. Solberg, Operations Research, John Wiley & Sons, 1987

[Romaniuk, 1993] - Romaniuk, S.T., L.O. Hall - Divide and Conquer Neural Networks, Neural Networks, Vol. 6, pp. 1105-1116, 1993

[Rosenblatt, 1958] - Rosenblatt F. - The perceptron: a probabilistic model for information storage and organisation in the brain, Psychological Review 65, pp. 386-408, 1958.

[Rosenblatt, 1962] - F. Rosenblatt, Principles of Neurodynamics, New York: Spartan, 1962

[Rumelhart, 1986] - D.E. Rumelhart, J.L. McClelland and the PDP research group, Parallel Distributed Processing, Explorations in the Microstructure of Cognition, The MIT Press, 1986

[Samad, 1990] - Samad, T. - Backpropagation improvements based on heuristic arguments. Proc. Int. Jont Conf. on Neural Networks vol. 1, 715-718 (1990).

[Sawyer, 1966] - W.W. Sawyer - A Path to Modern Mathematics, Penguin Books, 1966, pp. 189

[Schwartz, 1990] - D.B.Schwartz, V.K. Salaman, S.A. Solla, J.S. Denker, Exhaustive Learning, Neural Computation 2, 374-385, 1990

[Sejnowski 1986] - Sejnowski T.J., Rosenberg C.R. - NETtalk: a parallel network that learns to read aloud, The Johns Hopkins University Electrical Engineering and Computer Science Technical Report, JHU/EECS-86/01, 1986.

[Sethi, 1990a] - I. Sethi, Entropy nets: From decision trees to neural networks, Proc. of IEEE, pp.1605-1613, 1990

[Sethi,1990b] - I.Sethi, M.Otten, Comparison between entropy nets and decision tree classifiers, IJCNN, vol.III, pp.41-46, 1990

[Sietsma, 1991] - J. Sietsma, R.J.F. Dow - Creating Artificial Neural Networks That Generalise, Neural Networks, vol. 4, pp. 67-79, 1991

[Silviotti, 1987] - Silviotti M.A., Mahowald M.A., Mead C.A. - Real-time visual computation using analogue CMOS processing arrays, Advanced Research in VLSI: Proceedings of the 1987 Stanford Conference, P. Losleben (Ed.), Cambridge, MA: MIT Press, pp. 295-312, 1987.

[Sim, 1990] - M.S. Sim, C.C. Guest, Modification of Backpropagation Networks for Complex-Valued Signal Processing in Frequency Domain, Proceedings of IJCNN, June 1990, pp. II 27-31

[Sirat, 1990] - Sirat, J.-A., J.-P. Nadal, Neural Trees: A new tool for classification. Preprint, Laboratoires d'Electronique Philips, Limeil-Brevannes, France

[Sontag, 1989] - E.D.Sontag, H.J.Sussman, Backpropagation separates when perceptrons do, Technical Report SYCON-88-12, Rutgers Center for Systems and Control, 1989

[Stassinopoulos, 1994] - Stassinopoulos D, Bak P., Alstrom P - Self-organisation in a simple brain model, Invited talk to WCNN '94, San Diego, California, WCNN Proceedings, pp. I4 - I26, 1994.

[Takahashi, 1993] - Takahashi Y. - Generalization and Approximation Capabilities of Multilayer Networks, Neural Computation, vol. 5, no. 1, pp. 132-139, 1993

[Tesauro, 1990] - Tesauro G. - Neurogammon Wins Computer Olympiad. Neural Computation 1, pp. 321-323, 1990

[Vogl, 1988] - Vogl, T.P. et.al. - Accelerating the convergence of the backpropagation method. Biol. Cybern. 59, 257-263 (1988).

[von der Malsburg, 1973] - von der Malsburg C. - Self-organisation of orientation sensitive cells in the striata cortex, Kybernetik 14, pp. 85-100.

[von Neumann, 1982] - von Neumann J. - "First draft of a report on the EDVAC", The origins of digital computers: selected papers, B. Randall (Ed.) Berlin, Springer (3-rd edition).

[Watrous, 1987] - R.L. Watrous, Learning Algorithms for Connectionist Networks: Applied Gradient Methods of Nonlinear Optimisation, IEEE First International Conference on Neural Networks, San Diego 1987,

[Weir, 1991] - M.K. Weir, A method for self-determination of adaptive learning rates in back-propagation, Neural Networks, vol. 4, no. 3, pp. 371-379, 1991

[Weir, 1993] - M.K. Weir, J.G. Polhill, A Neural Implementation of Mitchell's Concept and Version Space Technique, Technical Report CS/93/12, Department of Computer Science, St Andrews University

[Weir, 1994] - M.K. Weir, J.G. Polhill, , Proceedings of IEEE International Conference of Neural Networks, 1994, vol. IV, pp. 2285-2290

[Weir, 1994] - M.K. Weir, J.G. Polhill, Implementing Mitchell's Concept and Version Spaces Technique in Neural Networks Using Weight Space Angle As The Partial Order Analogue, Technical Report CS/93/10 , Department of Computer Science, St Andrews University

[Weir, 1995] - M.K. Weir, S. Draghici, A backpropagation artificial neural network using complex weights, Technical Report CS/95/4, Department of Computer Science, St Andrews University

[Wender, 1994] - Wender A.J., Sherman J.H., Luciano D.S. - Human Physiology. The mechanisms of body function, McGraw Hill, 1994.

[Weymaere, 1991] - N. Weymaere, J. Martens, A Fast and Robust Learning Algorithm for Feedforward Neural Networks, Neural Networks, vol. 4, pp. 361-369, 1991

[Widrow, 1960] - B. Widrow, M.E. Hoff, Adaptive Switching Circuits, IRE WESCON Convention Record, part 4, 96-104, New York: IRE, 1960

[Widrow, 1975] - B. Widrow, J. McCool, M. Ball, The complex LMS algorithm, Proceedings of the IEEE, April 1975

[Wilensky,1990] - Wilensky, G.D., J.A. Neuhaus - Scaling of back-propagation training time to large dimensions, IJCNN, vol.III, pp.239-244, 1990

[Willshaw, 1969] - Willshaw D.J., Buneman O.P., Longuet-Higgins H.C. - Non-holographic associative memory, Nature 222:960-962, 1969

[Xu, 1992] - Xu L., S.Klasa and A. Yuille - Recent advances on techniques of static feedforward networks with supervised learning. IJNS Vol. 3 No. 3 253-290 (1992).

[Yamada, 1990] - K. Yamada, Handwritten numeral recognition by multilayered neural networks with improved learning algorithm, in Proc. Int. Joint Conf. on Neural Networks, San Diego, vol. II, pp. 7-12, 1990



# Appendix 1

## Statistical analysis of experimental data

The paired two-sample t-test for means was used to check the following hypothesis:

- i) The effect of the locking detection mechanism on the number of operations needed is positively significant in the sense that the locking detection mechanism leads the algorithm to a significantly faster training than that of the standard version.
- ii) The effect of the locking detection mechanism on the number of hyperplanes used in the solution is not significant.
- iii) The effect of the redundancy elimination mechanism on the number of hyperplanes used in the solution is positively significant in the sense that the redundancy mechanism determines the algorithm to find solutions with a number of hyperplanes significantly lower than the standard version.
- iv) The effect of the redundancy elimination mechanism on the training speed (number of operations) is not significant.

In order to check the hypothesis i) and iii) the calculated t-value was compared with the critical value for a one-tail test because one expects the values analysed to be consistently lower than the values obtained with the standard algorithm. For testing hypothesis ii) and iv), the calculated t-value was compared with the critical value for a two-tail test because the particular enhancement studied could have affected the results in either way (values significantly lower or significantly higher than the values obtained for the standard algorithm).

The following notation has been used:

hp - number of hyperplanes

counter - number of operations used to build the solution

red - the algorithm with the redundancy detection mechanism

lock - the algorithm with the locking detection mechanism

no red - the algorithm without the redundancy detection mechanism

no lock - the algorithm without the locking detection mechanism

The 2-grid problem with 20 patterns.

hp with red	hp no red			
9	9			
13	13			
12	13	t-Test: Paired Two-Sample for Means		
11	14		hp with red	hp no red
7	7	Mean	11.05	12.45
11	15	Variance	2.57631579	5.41842105
9	13	Observations	20	20
11	12	Pearson Correlation	0.59939076	
12	12	Pooled Variance	2.23947368	
13	13	Hypothesized Mean Difference	0	
10	17	df	19	
11	11	t	<b>-3.339116</b>	
9	9	P(T<=t) one-tail	0.00172375	
11	12	t Critical one-tail	<b>1.7291313</b>	
13	16	P(T<=t) two-tail	0.0034475	
12	12	t Critical two-tail	2.0930247	
11	12			
13	14			
11	12			
12	13			

This test shows that the average number of hyperplanes used by the standard algorithm is larger than the average number of hyperplanes used by the algorithm with the redundancy elimination mechanism (calculated t is greater in absolute value than the critical t for a one-side test). In other words, the redundancy elimination mechanism has brought a significant improvement by reducing the number of hyperplanes used.

counter lock	counter no lock			
1124325	1125537			
1141680	1142091			
590589	986166	t-Test: Paired Two-Sample for Means		
1249032	1485609		counter lock	counter no lock
676347	675846	Mean	1038254	1149094
1377804	1378764	Variance	5.08E+10	4.65E+10
1082223	982497	Observations	20	20
1112475	1112553	Pearson Correlation	0.713274	
1133094	1136217	Pooled Variance	3.47E+10	
1136322	1138983	Hypothesized Mean Difference	0	
1177968	1538388	df	19	
595206	1024110	t	<b>-2.96367</b>	
723966	776280	P(T<=t) one-tail	0.003989	
929373	1228575	t Critical one-tail	<b>1.729131</b>	
1090233	1445370	P(T<=t) two-tail	0.007977	
955617	994602	t Critical two-tail	2.093025	
1025412	1170399			
1213290	1208397			
1196298	1198524			
1233828	1232976			

This test shows that the average number of operations used by the standard algorithm is larger than the average number of operation used by the algorithm with the locking detection mechanism (calculated t is greater in absolute value than the critical t for a one-side test). In other words, the locking detection mechanism has brought a significant improvement by reducing the number of operations used.

counter red	counter no red			
1124325	1125630			
1141680	1140975			
590589	593127	t-Test: Paired Two-Sample for Means		
1249032	1270890		counter red	counter no red
676347	681477	Mean	1038254	1056214
1377804	1378068	Variance	5.08E+10	6.33E+10
1082223	984174	Observations	20	20
1112475	1111689	Pearson Correlation	0.94173	
1133094	1135278	Pooled Variance	5.34E+10	
1136322	1141347	Hypothesized Mean Difference	0	
1177968	1542357	df	19	
595206	593022	t	<b>-0.94056</b>	
723966	724614	P(T<=t) one-tail	0.179367	
929373	961509	t Critical one-tail	1.729131	
1090233	1087545	P(T<=t) two-tail	0.358735	
955617	958560	t Critical two-tail	<b>2.093025</b>	
1025412	1027773			
1213290	1206453			
1196298	1227372			
1233828	1232427			

This test shows that the null hypothesis (that the two populations have the same mean) cannot be rejected (calculated t is less in absolute value than the critical t for a two-side test). In other words, the redundancy elimination mechanism has not affected the training time.

hp lock	hp no lock			
9	9			
13	13			
12	12	t-Test: Paired Two-Sample for Means		
11	8		hp lock	hp no lock
7	7	Mean	11.05	11
11	11	Variance	2.576316	3.052632
9	12	Observations	20	20
11	11	Pearson Correlation	0.807009	
12	12	Pooled Variance	2.263158	
13	13	Hypothesized Mean Difference	0	
10	11	df	19	
11	11	t	<b>0.212946</b>	
9	8	P(T<=t) one-tail	0.416819	
11	11	t Critical one-tail	1.729131	
13	13	P(T<=t) two-tail	0.833637	
12	11	t Critical two-tail	<b>2.093025</b>	
11	11			
13	13			
11	11			
12	12			

This test shows that the null hypothesis (that the two populations have the same mean) cannot be rejected (calculated t is less in absolute value than the critical t for a two-side test). In other words, the locking detection mechanism has not affected the solution (the number of hyperplanes used).

The 2 spiral problem with 770 patterns.

counter lock	counter no lock			
293417427	923129910			
382352829	977966412			
148645659	834115119	t-Test: Paired Two-Sample for Means		
168468279	868718259		<i>counter lock</i>	<i>counter no lock</i>
121107651	867438783	Mean	177714120	869270000
121351542	838530270	Variance	8.7394E+15	6.0162E+15
147144717	859605570	Observations	16	16
170745660	852531663	Pearson Correlation	0.8133659	
103893951	772606191	Pooled Variance	5.8978E+15	
79117371	777396012	Hypothesized Mean Difference	0	
95639904	768584178	df	15	
141357318	840488607	t	<b>-50.8437</b>	
99336486	938855517	P(T<=t) one-tail	1.6406E-18	
175582266	864558651	t Critical one-tail	<b>1.753051</b>	
229094661	856555005	P(T<=t) two-tail	3.2813E-18	
366170205	1067239848	t Critical two-tail	2.13145086	

This test shows that the average number of operations used by the standard algorithm is larger than the average number of operations used by the algorithm with the locking detection mechanism (calculated t is greater in absolute value than the critical t for a one-side test). In other words, the locking detection mechanism has brought a significant improvement by reducing the number of operations used.

hp red	hp no red			
43	63			
94	194			
104	191	t-Test: Paired Two-Sample for Means		
133	242		<i>hp red</i>	<i>hp no red</i>
124	240	Mean	99.1875	186.5
122	241	Variance	524.029167	2212.93333
127	223	Observations	16	16
88	155	Pearson Correlation	0.91441227	
85	170	Pooled Variance	984.7	
78	170	Hypothesized Mean Difference	0	
113	178	df	15	
90	149	t	<b>-12.6061</b>	
98	211	P(T<=t) one-tail	1.0991E-09	
113	237	t Critical one-tail	<b>1.753051</b>	
81	165	P(T<=t) two-tail	2.1982E-09	
94	155	t Critical two-tail	2.13145086	

This test shows that the average number of hyperplanes used by the standard algorithm is larger than the average number of hyperplanes used by the algorithm with the redundancy elimination mechanism (calculated t is greater in absolute value than the critical t for a one-side test). In other words, the redundancy elimination mechanism has brought a significant improvement by reducing the number of hyperplanes used.

hp lock	hp no lock			
43	46			
94	93			
104	104	t-Test: Paired Two-Sample for Means		
133	114		<i>hp lock</i>	<i>hp no lock</i>
124	134	Mean	99.1875	102.4375
122	126	Variance	524.029167	474.395833
127	122	Observations	16	16
88	96	Pearson Correlation	0.90396463	
85	98	Pooled Variance	450.7125	
78	93	Hypothesized Mean Difference	0	
113	109	df	15	
90	95	t	<b>-1.31995</b>	
98	99	P(T<=t) one-tail	0.10332095	
113	135	t Critical one-tail	1.75305104	
81	89	P(T<=t) two-tail	0.20664189	
94	86	t Critical two-tail	<b>2.131451</b>	

This test shows that the null hypothesis (that the two populations have the same mean) cannot be rejected (calculated t is less in absolute value than the critical t for a two-side test). In other words, the locking detection mechanism has not affected the solution (the number of hyperplanes used).



counter red	counter no red			
293417427	320667798			
382352829	314732976			
148645659	172111338	t-Test: Paired Two-Sample for Means		
168468279	184248096		counter red	counter no red
121107651	136984230	Mean	177714120	188664488
121351542	149881524	Variance	8.7394E+15	8.1047E+15
147144717	120320223	Observations	16	16
170745660	315873459	Pearson Correlation	0.79829005	
103893951	85571115	Pooled Variance	6.7185E+15	
79117371	136411620	Hypothesized Mean Difference	0	
95639904	95600781	df	15	
141357318	148425783	t	<b>-0.7504</b>	
99336486	167850978	P(T<=t) one-tail	0.23231209	
175582266	197043258	t Critical one-tail	1.75305104	
229094661	105646173	P(T<=t) two-tail	0.46462419	
366170205	367262463	t Critical two-tail	<b>2.131451</b>	

This test shows that the null hypothesis (that the two populations have the same mean) cannot be rejected (calculated t is less in absolute value than the critical t for a two-side test). In other words, the redundancy elimination mechanism has not affected the training time.

## Appendix 2

### Estimating the locking tolerance

A method to estimate the locking tolerance will be given in the following. The method assumes that the aim of the algorithm is to separate patterns from two classes. The method will be presented for the 2D case but can be extended to more dimensions.

Let us suppose that the patterns from both classes are contained in a circle of radius  $D$ . This circle is assumed to be the smallest circle containing all patterns and it is assumed to be centred in the origin. Let us assume that the minimum distance between two patterns (of any class) is  $2d$ . The slope tolerance can be calculated as (see fig. 41):

$$\text{slope\_tolerance} = 2\alpha = 2 \arctan \frac{d}{D}$$

The justification of this formula is that we are not interested in positioning a dividing hyperplane with a better precision because an adjustment within this tolerance is not probable to change the classification of more than one pattern. Of course, the patterns could be arranged in radial groups in which case, an infinitesimal change of the position of the hyperplane could change the classification of many patterns but this extremely unfavourable situation is assumed not to be frequent. However, as discussed, the locking tolerance is not a critical parameter.

The locking tolerance can be estimated using a similar reasoning for the case in which the patterns are not distributed around the origin but in an arbitrary position.

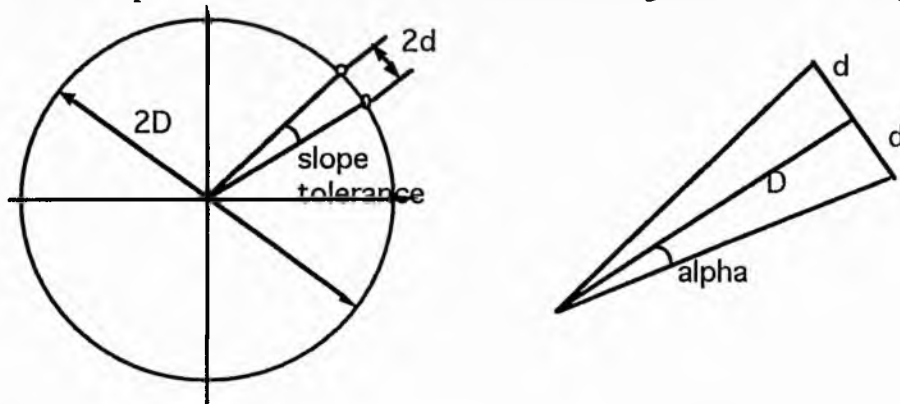


Fig. 41 Estimating the locking tolerance. All patterns of both classes are contained in the circle.

## Appendix 3

### Equations of complex backpropagation in radial/phase terms with two independent real activation functions

This approach uses two independent real activation functions<sup>1</sup> at the level of each neuron. The approach uses the radius/phase form of a complex number. The excitation of a neuron is calculated as the complex sum of the incoming excitations. A complex weight has a radial component and a phase component. The effect of a weight is to change separately the radius and the phase of the complex value it acts upon (the multiplication between the weight and the value is not performed as between two complex numbers).

A complex weight (radius and phase) performs a rotation (the phase weight will affect the angle) and a scaling (the radius weight will affect the magnitude) in the Argand plane (the complex plane) as shown in fig. 1.

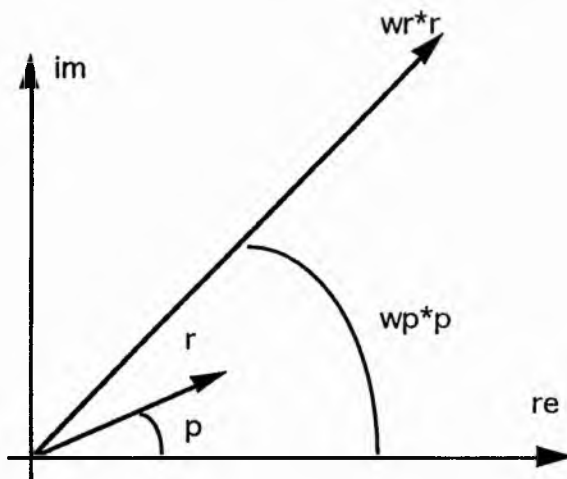


Fig. 1. The effect of a weight (radius =  $w_r$  and phase =  $w_p$ ) upon a  $(r, p)$  pair.

---

<sup>1</sup>Real functions of a real variable i.e.  $f: \mathbb{R} \rightarrow \mathbb{R}$

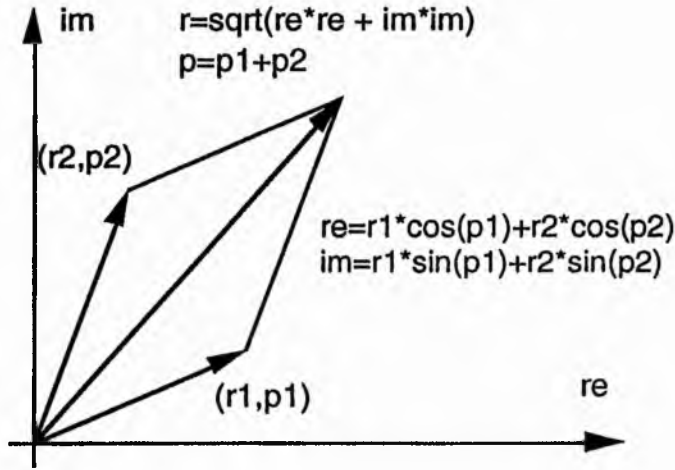


Fig. 2. What a neuron (with two incoming links) does.  $(r_1, p_1)$  and  $(r_2, p_2)$  are the incoming excitation i.e. each is already affected by its corresponding weight as in fig.1.

At the level of the neuron, the radius and phase parts are kept apart by using two different activation functions. Each of this function is a real function of a real variable.

Such a network is not equivalent to a pair of real networks and is equivalent with any other complex network which uses two independent real activation functions.

### The derivation of the gradient descent equations in radial and phase terms

The approach uses two real functions of a real variable as the radius and phase activation functions of a neuron:

$$\begin{cases} \text{out}^{\text{ra}} = \sigma_{\text{ra}}(ex^{\text{ra}}) \\ \text{out}^{\text{p}} = \sigma_{\text{p}}(ex^{\text{p}}) \end{cases} \quad (1)$$

The total real and imaginary excitations of a neuron are calculated by adding the real and imaginary components (the x and y components) of the incoming excitations:

$$\begin{cases} ex_k^x = \sum_{j=1}^n ex_j^x = \sum_{j=1}^n w_{jk}^{\text{ra}} r_j \cos(w_{jk}^{\text{p}} \theta_j) \\ ex_k^y = \sum_{j=1}^n ex_j^y = \sum_{j=1}^n w_{jk}^{\text{ra}} r_j \sin(w_{jk}^{\text{p}} \theta_j) \end{cases} \quad (2)$$

The radial and phase components are then calculated from the real and imaginary components:

$$\begin{cases} ex_k^{ra} = \sqrt{(ex_k^x)^2 + (ex_k^y)^2} \\ ex_k^p = \arctg\left(\frac{ex_k^y}{ex_k^x}\right) \end{cases} \quad (3)$$

The error is defined in radial and phase terms as the squared difference between the actual output values and the targets:

$$E(w_{jk}^{ra}, w_{jk}^p) = \frac{1}{2} \sum_k \left[ (out_k^{ra} - targ_k^{ra})^2 + (out_k^p - targ_k^p)^2 \right] \quad (4)$$

The gradient of the error-weight surface has a radial component and a phase component. The radial component for the output units can be calculated as:

$$\frac{\partial E}{\partial w_{jk}^{ra}} = \frac{\partial E}{\partial out_k^{ra}} \frac{\partial out_k^{ra}}{\partial ex_k^{ra}} \frac{\partial ex_k^{ra}}{\partial w_{jk}^{ra}} + \frac{\partial E}{\partial out_k^p} \frac{\partial out_k^p}{\partial ex_k^p} \frac{\partial ex_k^p}{\partial w_{jk}^{ra}} \quad (5)$$

in which:

$$\frac{\partial E}{\partial out_k^{ra}} = (out_k^{ra} - targ_k^{ra}) \quad (6)$$

$$\frac{\partial E}{\partial out_k^p} = (out_k^p - targ_k^p) \quad (7)$$

and

$$\frac{\partial out_k^{ra}}{\partial ex_k^{ra}} = \sigma'_{ra} \quad (8)$$

$$\frac{\partial out_k^p}{\partial ex_k^p} = \sigma'_p \quad (9)$$

The last factors in each term of the expression (5) of the radial component of the gradient can be calculated as:

$$\frac{\partial ex_k^{ra}}{\partial w_{jk}^{ra}} = \frac{\partial ex_k^{ra}}{\partial ex_k^x} \frac{\partial ex_k^x}{\partial w_{jk}^{ra}} + \frac{\partial ex_k^{ra}}{\partial ex_k^y} \frac{\partial ex_k^y}{\partial w_{jk}^{ra}} \quad (10)$$

From (3) it follows that:

$$\frac{\partial ex_k^{ra}}{\partial ex_k^x} = \frac{1}{2} \frac{2ex_k^x}{\sqrt{(ex_k^x)^2 + (ex_k^y)^2}} = \cos ex_k^p \quad (11)$$

$$\frac{\partial ex_k^x}{\partial w_{jk}^{ra}} = r_j \cos w_{jk}^p \theta_j \quad (12)$$

$$\frac{\partial ex_k^{ra}}{\partial ex_k^y} = \frac{1}{2} \frac{2ex_k^y}{\sqrt{(ex_k^x)^2 + (ex_k^y)^2}} = \sin ex_k^p \quad (13)$$

$$\frac{\partial ex_k^y}{\partial w_{jk}^{ra}} = r_j \sin w_{jk}^p \theta_j \quad (14)$$

Using (11), (12), (13) and (14), one can write (10) as:

$$\frac{\partial ex_k^{ra}}{\partial w_{jk}^{ra}} = r_j (\cos w_{jk}^p \theta_j \cos ex_k^p + \sin w_{jk}^p \theta_j \sin ex_k^p) \quad (15)$$

$$\frac{\partial ex_k^{ra}}{\partial w_{jk}^{ra}} = r_j \cos(w_{jk}^p \theta_j - ex_k^p) \quad (16)$$

The derivative of the phase excitation with respect to the radial weight can be calculated in a similar fashion:

$$\frac{\partial ex_k^p}{\partial w_{jk}^{ra}} = \frac{\partial ex_k^p}{\partial ex_k^x} \frac{\partial ex_k^x}{\partial w_{jk}^{ra}} + \frac{\partial ex_k^p}{\partial ex_k^y} \frac{\partial ex_k^y}{\partial w_{jk}^{ra}} \quad (17)$$

From (3):

$$\frac{\partial ex_k^p}{\partial ex_k^x} = \frac{1}{1 + \left(\frac{ex_k^y}{ex_k^x}\right)^2} \frac{-ex_k^y}{(ex_k^x)^2} = \frac{-ex_k^y}{(ex_k^x)^2 + (ex_k^y)^2} = \frac{-ex_k^y}{(ex_k^{ra})^2} = \frac{-\sin ex_k^p}{ex_k^{ra}} \quad (18)$$

$$\frac{\partial ex_k^x}{\partial w_{jk}^{ra}} = r_j \cos w_{jk}^p \theta_j \quad (19)$$

$$\frac{\partial ex_k^p}{\partial ex_k^y} = \frac{1}{1 + \left(\frac{ex_k^y}{ex_k^x}\right)^2} \frac{1}{ex_k^x} = \frac{ex_k^x}{(ex_k^x)^2 + (ex_k^y)^2} = \frac{ex_k^x}{(ex_k^{ra})^2} = \frac{\cos ex_k^p}{ex_k^{ra}} \quad (20)$$

$$\frac{\partial ex_k^y}{\partial w_{jk}^{ra}} = r_j \sin w_{jk}^p \theta_j \quad (21)$$

Using the expressions (18), (19), (20) and (21), one can write (17) as follows:

$$\frac{\partial ex_k^p}{\partial w_{jk}^{ra}} = r_j \left( \cos w_{jk}^p \theta_j \frac{-\sin ex_k^p}{ex_k^{ra}} + \sin w_{jk}^p \theta_j \frac{\cos ex_k^p}{ex_k^{ra}} \right) \quad (22)$$

$$\frac{\partial ex_k^p}{\partial w_{jk}^{ra}} = \frac{r_j}{ex_k^{ra}} \sin(w_{jk}^p \theta_j - ex_k^p) \quad (23)$$

Now, the radial component of the gradient of the error-weight surface for the output units (5) can be written using (16) and (23):

$$\frac{\partial E}{\partial w_{jk}^{ra}} = \delta_k^{ra} r_j \cos(w_{jk}^p \theta_j - ex_k^p) + \delta_k^p \frac{r_j}{ex_k^{ra}} \sin(w_{jk}^p \theta_j - ex_k^p) \quad (24)$$

where:

$$\begin{cases} \delta_k^{ra} = \frac{\partial E}{\partial out_k^{ra}} \frac{\partial out_k^{ra}}{\partial ex_k^{ra}} \\ \delta_k^p = \frac{\partial E}{\partial out_k^p} \frac{\partial out_k^p}{\partial ex_k^p} \end{cases} \quad (25)$$

$$\begin{cases} \delta_k^{ra} = (out_k^{ra} - targ_k^{ra}) \sigma_{ra}' \\ \delta_k^p = (out_k^p - targ_k^p) \sigma_p' \end{cases}$$

The phase component of the gradient of the error-weight surface can be calculated as:

$$\frac{\partial E}{\partial w_{jk}^p} = \frac{\partial E}{\partial out_k^{ra}} \frac{\partial out_k^{ra}}{\partial ex_k^{ra}} \frac{\partial ex_k^{ra}}{\partial w_{jk}^p} + \frac{\partial E}{\partial out_k^p} \frac{\partial out_k^p}{\partial ex_k^p} \frac{\partial ex_k^p}{\partial w_{jk}^p} \quad (26)$$

in which:

$$\frac{\partial ex_k^{ra}}{\partial w_{jk}^p} = \frac{\partial ex_k^{ra}}{\partial ex_k^x} \frac{\partial ex_k^x}{\partial w_{jk}^p} + \frac{\partial ex_k^{ra}}{\partial ex_k^y} \frac{\partial ex_k^y}{\partial w_{jk}^p} \quad (27)$$

From (3) and (2):

$$\frac{\partial ex_k^{ra}}{\partial ex_k^x} = \frac{1}{2} \frac{2ex_k^x}{\sqrt{(ex_k^x)^2 + (ex_k^y)^2}} = \cos ex_k^p \quad (28)$$

$$\frac{\partial ex_k^x}{\partial w_{jk}^p} = w_{jk}^{ra} r_j (-\sin w_{jk}^p \theta_j) \theta_j \quad (29)$$

$$\frac{\partial ex_k^{ra}}{\partial ex_k^x} = \frac{1}{2} \frac{2ex_k^y}{\sqrt{(ex_k^x)^2 + (ex_k^y)^2}} = \sin ex_k^p \quad (30)$$

$$\frac{\partial ex_k^y}{\partial w_{jk}^p} = w_{jk}^{ra} r_j (\cos w_{jk}^p \theta_j) \theta_j \quad (31)$$

The expression (27) becomes:

$$\frac{\partial ex_k^{ra}}{\partial w_{jk}^p} = w_{jk}^{ra} r_j \theta_j (\sin ex_k^p \cos w_{jk}^p \theta_j - \cos ex_k^p \sin w_{jk}^p \theta_j) \quad (32)$$

$$\frac{\partial ex_k^{ra}}{\partial w_{jk}^p} = w_{jk}^{ra} r_j \theta_j \sin(ex_k^p - w_{jk}^p \theta_j) \quad (33)$$

Similarly, the derivative of the phase excitation with respect to the phase weight can be calculated:

$$\frac{\partial ex_k^p}{\partial w_{jk}^p} = \frac{\partial ex_k^p}{\partial ex_k^x} \frac{\partial ex_k^x}{\partial w_{jk}^p} + \frac{\partial ex_k^p}{\partial ex_k^y} \frac{\partial ex_k^y}{\partial w_{jk}^p} \quad (34)$$

where each factor is calculated from (2) and (3) again:

$$\frac{\partial ex_k^p}{\partial ex_k^x} = \frac{1}{1 + \left(\frac{ex_k^y}{ex_k^x}\right)^2} \frac{-ex_k^y}{(ex_k^x)^2} = \frac{-ex_k^y}{(ex_k^x)^2 + (ex_k^y)^2} = \frac{-ex_k^y}{(ex_k^{ra})^2} = \frac{-\sin ex_k^p}{ex_k^{ra}} \quad (35)$$

$$\frac{\partial ex_k^x}{\partial w_{jk}^p} = w_{jk}^{ra} r_j (-\sin w_{jk}^p \theta_j) \theta_j \quad (36)$$

$$\frac{\partial ex_k^p}{\partial ex_k^y} = \frac{1}{1 + \left(\frac{ex_k^y}{ex_k^x}\right)^2} \frac{1}{ex_k^x} = \frac{ex_k^x}{(ex_k^x)^2 + (ex_k^y)^2} = \frac{ex_k^x}{(ex_k^{ra})^2} = \frac{\cos ex_k^p}{ex_k^{ra}} \quad (37)$$



$$\frac{\partial ex_k^y}{\partial w_{jk}^p} = w_{jk}^{ra} r_j \cos w_{jk}^p \theta_j \cdot \theta_j \quad (38)$$

Thus, (34) becomes:

$$\frac{\partial ex_k^p}{\partial w_{jk}^p} = \frac{w_{jk}^{ra} r_j \theta_j}{ex_k^{ra}} (\cos w_{jk}^p \theta_j \cos ex_k^p + \sin w_{jk}^p \theta_j \sin ex_k^p) \quad (39)$$

$$\frac{\partial ex_k^p}{\partial w_{jk}^p} = \frac{w_{jk}^{ra} r_j \theta_j}{ex_k^{ra}} \cos(ex_k^p - w_{jk}^p \theta_j) \quad (40)$$

Now, expression (26) which is the phase component of the gradient of the error-weight surface can be written as:

$$\frac{\partial E}{\partial w_{jk}^p} = \delta_k^{ra} r_j w_{jk}^{ra} \theta_j \sin(-w_{jk}^p \theta_j + ex_k^p) + \delta_k^p \frac{r_j}{ex_k^{ra}} w_{jk}^{ra} \theta_j \cos(w_{jk}^p \theta_j - ex_k^p) \quad (41)$$

where the delta values are those in (25)

The delta values for the hidden units can be calculated as in the following (see fig. 3).

$$\delta_j^{ra} = \frac{\partial E}{\partial out_j^{ra}} \frac{\partial out_j^{ra}}{\partial ex_j^{ra}} \quad (42)$$

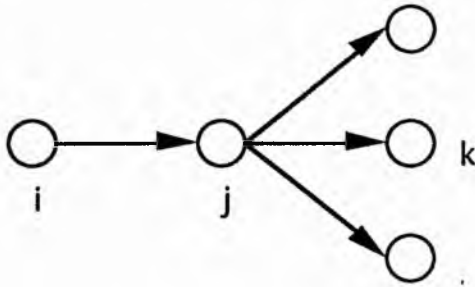


Fig. 3. Calculating the delta values for hidden units.

$$\begin{aligned} \frac{\partial E}{\partial out_j^{ra}} &= \sum_k \left( \frac{\partial E}{\partial out_k^{ra}} \frac{\partial out_k^{ra}}{\partial out_j^{ra}} + \frac{\partial E}{\partial out_k^p} \frac{\partial out_k^p}{\partial out_j^{ra}} \right) = \\ &= \sum_k \left( \frac{\partial E}{\partial out_k^{ra}} \frac{\partial out_k^{ra}}{\partial ex_k^{ra}} \frac{\partial ex_k^{ra}}{\partial out_j^{ra}} + \frac{\partial E}{\partial out_k^{ra}} \frac{\partial out_k^{ra}}{\partial ex_k^p} \frac{\partial ex_k^p}{\partial out_j^{ra}} + \frac{\partial E}{\partial out_k^p} \frac{\partial out_k^p}{\partial ex_k^p} \frac{\partial ex_k^p}{\partial out_j^{ra}} + \frac{\partial E}{\partial out_k^p} \frac{\partial out_k^p}{\partial ex_k^{ra}} \frac{\partial ex_k^{ra}}{\partial out_j^{ra}} \right) \end{aligned} \quad (43)$$

As the radial and phase activation functions of a neuron are independent, the radial output does not depend on the phase excitation and vice-versa. Therefore, the second and fourth terms in (43) are zero and the expression becomes:

$$\frac{\partial E}{\partial out_j^{ra}} = \sum_k \left( \frac{\partial E}{\partial out_k^{ra}} \frac{\partial out_k^{ra}}{\partial ex_k^{ra}} \frac{\partial ex_k^{ra}}{\partial out_j^{ra}} + \frac{\partial E}{\partial out_k^p} \frac{\partial out_k^p}{\partial ex_k^p} \frac{\partial ex_k^p}{\partial out_j^{ra}} \right) \quad (44)$$

But

$$\frac{\partial E}{\partial out_k^{ra}} \frac{\partial out_k^{ra}}{\partial ex_k^{ra}} = \delta_k^{ra} \quad \text{and} \quad (45)$$

$$\frac{\partial E}{\partial out_k^p} \frac{\partial out_k^p}{\partial ex_k^{ra}} = \delta_k^p \quad \text{and we need to calculate only the last factors in each term of (44).}$$

$$\frac{\partial ex_k^{ra}}{\partial out_j^{ra}} = \frac{\partial ex_k^{ra}}{\partial r_j} = \frac{\partial ex_k^{ra}}{\partial ex_k^x} \frac{\partial ex_k^x}{\partial r_j} + \frac{\partial ex_k^{ra}}{\partial ex_k^y} \frac{\partial ex_k^y}{\partial r_j} \quad (46)$$

By differentiating (2) with respect to the radial excitation  $r_j$ , one obtains:

$$\frac{\partial ex_k^x}{\partial r_j} = w_{jk}^{ra} \cos w_{jk}^p \theta_j \quad \text{and} \quad (47)$$

$$\frac{\partial ex_k^y}{\partial r_j} = w_{jk}^{ra} \sin w_{jk}^p \theta_j \quad (48)$$

Using (11), (47), (13) and (48), expression (46) can be written as:

$$\frac{\partial ex_k^{ra}}{\partial out_j^{ra}} = w_{jk}^{ra} (\cos w_{jk}^p \theta_j \cos ex_k^p + \sin w_{jk}^p \theta_j \sin ex_k^p) = w_{jk}^{ra} \cos(w_{jk}^p \theta_j - ex_k^p) \quad (49)$$

The last factor in the second term of (44) is:

$$\frac{\partial ex_k^p}{\partial out_j^{ra}} = \frac{\partial ex_k^p}{\partial r_j} = \frac{\partial ex_k^p}{\partial ex_k^x} \frac{\partial ex_k^x}{\partial r_j} + \frac{\partial ex_k^p}{\partial ex_k^y} \frac{\partial ex_k^y}{\partial r_j} \quad (50)$$

Using (18), (47), (20) and (48) one can rewrite (50) as:

$$\frac{\partial ex_k^p}{\partial out_j^{ra}} = \frac{w_{jk}^{ra}}{ex_k^{ra}} \sin(w_{jk}^p \theta_j - ex_k^p) \quad (51)$$

Using (49) and (51), expression (44) can be written as:

$$\frac{\partial E}{\partial out_j^{ra}} = \sum_k \left( \delta_k^{ra} w_{jk}^{ra} \cos(w_{jk}^p \theta_j - ex_k^p) + \delta_k^p \frac{w_{jk}^{ra}}{ex_k^{ra}} \sin(w_{jk}^p \theta_j - ex_k^p) \right) \quad (52)$$

And finally, expression (42) which is the radial delta value for a hidden unit can be written as a function of the delta values of the output units:

$$\delta_j^{ra} = \sigma'_{ra} \sum_k \left( \delta_k^{ra} w_{jk}^{ra} \cos(w_{jk}^p \theta_j - ex_k^p) + \delta_k^p \frac{w_{jk}^{ra}}{ex_k^{ra}} \sin(w_{jk}^p \theta_j - ex_k^p) \right) \quad (53)$$

A similar expression can be obtained for the phase delta value of the hidden unit  $j$ . In order to obtain this expression, we must calculate:

$$\delta_j^{ra} = \frac{\partial E}{\partial out_j^p} \frac{\partial out_j^p}{\partial ex_j^p} \quad (54)$$

The derivative of the error with respect to the phase output of hidden unit  $j$  is:

$$\frac{\partial E}{\partial out_j^p} = \sum_k \left( \frac{\partial E}{\partial out_k^{ra}} \frac{\partial out_k^{ra}}{\partial ex_k^{ra}} \frac{\partial ex_k^{ra}}{\partial out_j^p} + \frac{\partial E}{\partial out_k^p} \frac{\partial out_k^p}{\partial ex_k^p} \frac{\partial ex_k^p}{\partial out_j^p} \right) \quad (55)$$

in which we have ignored the terms which are zero as in (43).

As the first two factors in each term form the delta values as in (45), we need to calculate the last factors in each term.

$$\frac{\partial ex_k^{ra}}{\partial out_j^p} = \frac{\partial ex_k^{ra}}{\partial \theta_j} = \frac{\partial ex_k^{ra}}{\partial ex_k^x} \frac{\partial ex_k^x}{\partial \theta_j} + \frac{\partial ex_k^{ra}}{\partial ex_k^y} \frac{\partial ex_k^y}{\partial \theta_j} \quad (56)$$

By differentiating (2) with respect to  $\theta_j$ , one obtains:

$$\frac{\partial ex_k^x}{\partial \theta_j} = w_{jk}^{ra} r_j (-\sin w_{jk}^p \theta_j) w_{jk}^p \quad (57)$$

$$\frac{\partial ex_k^y}{\partial \theta_j} = w_{jk}^{ra} r_j \cos w_{jk}^p \theta_j \cdot w_{jk}^p \quad (58)$$

Using (11), (57), (13) and (58), expression (56) can be put into the following form:

$$\frac{\partial ex_k^{ra}}{\partial out_j^p} = w_{jk}^{ra} r_j w_{jk}^p \sin(ex_k^p - w_{jk}^p \theta_j) \quad (59)$$

The last factor in the last term of (55) can be calculated as:

$$\frac{\partial ex_k^p}{\partial out_j^p} = \frac{\partial ex_k^p}{\partial \theta_j} = \frac{\partial ex_k^p}{\partial ex_k^x} \frac{\partial ex_k^x}{\partial \theta_j} + \frac{\partial ex_k^p}{\partial ex_k^y} \frac{\partial ex_k^y}{\partial \theta_j} \quad (60)$$

Using (18), (57), (20) and (58) one can write expression (60) in the following form:

$$\frac{\partial ex_k^p}{\partial out_j^p} = \frac{\partial ex_k^p}{\partial \theta_j} = \frac{w_{jk}^{ra} r_j w_{jk}^p}{ex_k^{ra}} \cos(ex_k^p - w_{jk}^p \theta_j) \quad (61)$$

And finally, the expression (54) can be written as:

$$\delta_k^p = \sigma_p' \sum_k \left[ \delta_k^{ra} w_{jk}^{ra} r_j w_{jk}^p \sin(-w_{jk}^p \theta_j + ex_k^p) + \delta_k^p \frac{w_{jk}^{ra}}{ex_k^{ra}} r_j w_{jk}^p \cos(w_{jk}^p \theta_j - ex_k^p) \right] \quad (62)$$

Although not as simple as the correspondent equations of the standard backpropagation, expressions (53) and (62) are the expressions which allow us to calculate the delta values of a hidden unit by backpropagating the delta values of the output units. Such recurrent relations can be applied to any number of layers.

The weight changes are given by the expressions:

$$\begin{cases} \Delta w_{jk}^{ra} = -\varepsilon_{ra} \frac{\partial E}{\partial w_{jk}^{ra}} \\ \Delta w_{jk}^p = -\varepsilon_p \frac{\partial E}{\partial w_{jk}^p} \end{cases} \quad (63)$$

In summary, the complex backpropagation in radius/phase terms and using two independent real functions of a real variable as the radius and phase activation functions of a neuron can be described by the following set of expressions.

The weights are changed according to:

$$\begin{cases} \Delta w_{jk}^{ra} = -\varepsilon_{ra} \frac{\partial E}{\partial w_{jk}^{ra}} \\ \Delta w_{jk}^p = -\varepsilon_p \frac{\partial E}{\partial w_{jk}^p} \end{cases} \quad (64)$$

where

$$\frac{\partial E}{\partial w_{jk}^{ra}} = \delta_k^{ra} r_j \cos(w_{jk}^p \theta_j - ex_k^p) + \delta_k^p \frac{r_j}{ex_k^{ra}} \sin(w_{jk}^p \theta_j - ex_k^p) \quad \text{and} \quad (65)$$

$$\frac{\partial E}{\partial w_{jk}^p} = \delta_k^{ra} r_j w_{jk}^{ra} \theta_j \sin(-w_{jk}^p \theta_j + ex_k^p) + \delta_k^p \frac{r_j}{ex_k^{ra}} w_{jk}^{ra} \theta_j \cos(w_{jk}^p \theta_j - ex_k^p) \quad (66)$$

The delta values in (62) and (63) are calculated using

$$\begin{cases} \delta_k^{ra} = (out_k^{ra} - targ_k^{ra}) \sigma_{ra}' \\ \delta_k^p = (out_k^p - targ_k^p) \sigma_p' \end{cases} \quad \text{for output units and} \quad (67)$$

$$\begin{cases} \delta_k^{ra} = \sigma_{ra}' \sum_k \left[ \delta_k^{ra} w_{jk}^{ra} \cos(w_{jk}^p \theta_j - ex_k^p) + \delta_k^p \frac{w_{jk}^{ra}}{ex_k^{ra}} \sin(w_{jk}^p \theta_j - ex_k^p) \right] \\ \delta_k^p = \sigma_p' \sum_k \left[ \delta_k^{ra} w_{jk}^{ra} r_j w_{jk}^p \sin(-w_{jk}^p \theta_j + ex_k^p) + \delta_k^p \frac{w_{jk}^{ra}}{ex_k^{ra}} r_j w_{jk}^p \cos(w_{jk}^p \theta_j - ex_k^p) \right] \end{cases} \quad (68)$$

for hidden units.

## **Appendix 4**

### **Details of some experiments with the complex backpropagation**

This appendix presents the details of the experiments investigating the training properties of the complex backpropagation. The main purpose of these experiments is to ensure that the complex backpropagation training algorithm and its implementation are correct.

This appendix is aimed at the reader who is interested in reproducing the results and/or further experimental investigations.

#### **1. Testing the training algorithm**

##### **General comments on the experimental set-up and results**

Most of the experiments were performed starting from five different initial weight states. The weights of these initial weight states were chosen randomly between -0.5 and 0.5.

The training was performed with a target error limit of 0.0000001. In certain cases, the training was stopped before this limit was reached and the error at the end of the training is reported. The error reported here is the maximum (over the set of patterns) absolute value of the difference between the target (radial or phase) and the output.

The number of epochs of each training session is reported (rounded to the nearest thousand) in order to allow a comparison between the difficulty of various training sessions.

Although this error limit appears to be a little extreme (much smaller than the normal error limits used in backpropagation training), such a small value is needed if a comparison of the weight states is sought. The justification for this is given in chapter 8, section 8.4.

## Experiments

### Experiment 1

The first experiment involves just one link between two neurons in the conditions in which no bias is used. The architecture is presented in fig. 6. As explained, a random weight state and 2 randomly chosen input patterns are used to generate a training set containing 2 complex I/O patterns.



Fig. 6 A one-link architecture, no bias.

Five training sessions were performed starting with five different initial weight states. The results are summarised in table 2.

initial weight state	original solution found	no. of epochs	error at the end of training
state 1	yes	3000	<0.00000001
state 2	did not converge	100000	>1.
state 3	yes	3000	<0.00000001
state 4	yes	3000	<0.00000001
state 5	did not converge	100000	>1.

Table 2. Testing the abilities of retrieving the original weight state for a 1-1 architecture without bias. The number of epochs is rounded to the nearest thousand.

The initial weight states used are presented in table 3.

initial weight state	radius weight	phase weight
state 1	0.013871	-0.324274
state 2	-0.191366	0.034532
state 3	0.447630	-0.328272
state 4	0.202231	-0.273583
state 5	-0.005234	-0.375301

Table 3. Initial weight states for experiment 1.

Other training parameters for experiment 2:

no. of patterns = 2

no. of inputs = 1

number of outputs = 1

error limit = 0.00000001

sigmoid's parameter = 0.5

radius learning rate = 0.1

phase learning rate = 0.1

momentum = 0.5

radius bias = 1.000

phase bias = 1.000

### Experiment 2.

The second experiment used a 1-1 architecture with bias. The architecture is presented in fig. 7.

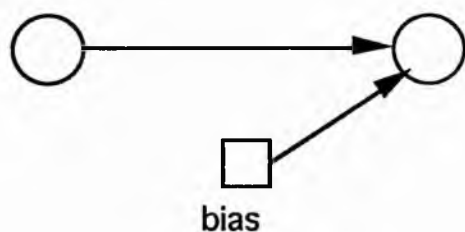


Fig. 7 A one-link architecture with bias.

The results are summarised in table 4.

initial weight state	original solution found	no. of epochs	error at the end of training
state 1	yes	30,000	>0.00000001
state 2	yes	79,000	>0.00000001
state 3	yes	24,000	>0.00000001
state 4	yes	24,000	>0.00000001
state 5	yes	31,000	>0.00000001

Table 4. Testing the abilities of retrieving the original weight state for a 1-1 architecture with bias. The number of epochs is rounded at the nearest thousand.

The initial weight states used are presented in table 5.



initial weight state	radius weight	phase weight	bias radius weight	bias phase weight
state 1	0.013871	-0.324274	0.154317	0.034532
state 2	-0.191366	0.034532	0.473815	-0.328272
state 3	0.447630	-0.328272	0.351115	-0.273583
state 4	0.202231	-0.273583	0.247383	-0.375301
state 5	-0.005234	-0.375301	0.041948	-0.110370

Table 5. Initial weight states for experiment 2.

Other training parameters for experiment 2:

no. of patterns = 2

no. of inputs = 1

number of outputs = 1

error limit = 0.00000001

sigmoid's parameter = 0.5

radius learning rate = 0.1

phase learning rate = 0.1

momentum = 0.5

radius bias = 1.000

phase bias = 1.000

### Experiment 3

A 111 architecture without bias is trained with four patterns generated with a 11 architecture with bias. The results of 5 training sessions are presented in table 6.

initial weight state	original solution found	number of epochs	error at the end of training
state 1	n/a	142,000	0.002527
state 2	n/a	156,000	4.942949
state 3	n/a	141,000	0.027793
state 4	n/a	142,000	5.0133726
state 5	n/a	141,000	3.288516

Table 6. Training an 111 architecture without bias with four patterns generated with a 11 architecture with bias. Only two training sessions were successful.

initial weight state	original solution found	number of epochs	error at the end of training
state 1	n/a	111,000	0.013596
state 2	n/a	111,000	0.00384
state 3	n/a	111,000	0.0121
state 4	n/a	111,000	0.01444
state 5	n/a	111,000	0.004146

Table 7. Training a 111 architecture with bias with the training set generated by a 11 architecture with bias. All training sessions were successful in the sense that a reasonably small error was obtained at the end of the training.

initial weight state	original solution found	number of epochs	error at the end of training
state 1	no	103,000	0.0127130
state 2	no	102,000	0.0115440
state 3	no	102,000	0.0207423
state 4	no	103,000	0.0058204
state 5	no	103,000	0.0068192

Table 8. Training a 111 architecture with bias with a pattern set containing four patterns generated with a different weight state but the same architecture.

Training parameters are as follows:

no. of patterns = 4

no. of inputs = 1

number of outputs = 1

error limit = 0.00000001

sigmoid's parameter = 0.5

radius learning rate = 0.01

phase learning rate = 0.00001

momentum = 0.5

radius bias = 1.000

phase bias = 1.000

initial weight state	original solution found	number of epochs	error at the end of training
state 1	no	145,000	0.00103
state 2	no	145,000	0.00717
state 3	no	145,000	0.00240
state 4	no	145,000	0.00173
state 5	no	145,000	0.00516

Table 9. Training a 111 architecture with bias with a pattern set containing four patterns generated with a different weight state but the same architecture. The only difference from the previous experiment is the learning rates.

Training parameters are as follows:

no. of patterns = 4

no. of inputs = 1

number of outputs = 1

error limit = 0.00000001

sigmoid's parameter = 0.5

radius learning rate = 0.1

phase learning rate = 0.1

momentum = 0.5

radius bias = 1.000

phase bias = 1.000

The initial weight states used for the 111 architecture are:

initial state 1:

node[1][0] from node[0][0] ra=0.013871,p=-0.324274

node[1][0] from node[0][1] ra=0.473815,p=-0.328272

node[2][0] from node[1][0] ra=-0.191366,p=0.034532

node[2][0] from node[1][1] ra=0.351115,p=-0.273583

initial state 2:

node[1][0] from node[0][0] ra=0.447630,p=-0.328272

node[1][0] from node[0][1] ra=0.247383,p=-0.375301

node[2][0] from node[1][0] ra=0.202231,p=-0.273583

node[2][0] from node[1][1] ra=0.041948,p=-0.110370

initial state 3:

node[1][0] from node[0][0] ra=-0.005234,p=-0.375301  
node[1][0] from node[0][1] ra=0.138615,p=-0.131947  
node[2][0] from node[1][0] ra=-0.416105,p=-0.110370  
node[2][0] from node[1][1] ra=0.491729,p=0.035386

initial state 4:

node[1][0] from node[0][0] ra=-0.222770,p=-0.131947  
node[1][0] from node[0][1] ra=0.382839,p=0.146474  
node[2][0] from node[1][0] ra=0.483459,p=0.035386  
node[2][0] from node[1][1] ra=0.383572,p=0.280236

initial state 5:

node[1][0] from node[0][0] ra=0.265679,p=0.146474  
node[1][0] from node[0][1] ra=0.411481,p=-0.348079  
node[2][0] from node[1][0] ra=0.267144,p=0.280236  
node[2][0] from node[1][1] ra=0.312738,p=-0.185324

The bias neuron appears as the last neuron on the previous layer (neuron [0][1] for the hidden layer and neuron [1][1] for the output layer).

initial weight state	original solution found	number of epochs	error at the end of training
state 1	no	752,000	0.0007429
state 2	no	753,000	0.0003170
state 3	no	752,000	0.0010554
state 4	no	754,000	0.0004930
state 5	no	752,000	0.0007809

Table 10. Training a 121 architecture with four patterns generated with a different weight state but the same architecture.

Training parameters are as follows:

no. of patterns = 4

no. of inputs = 1

number of outputs = 1

error limit = 0.00000001

sigmoid's parameter = 0.5

radius learning rate = 0.1

phase learning rate = 0.1

momentum = 0.5

radius bias = 1.000

phase bias = 1.000

A different training session in which the learning rates were varied managed to match the 0.00000001 error limit. However, the solution weight state was not the original weight state.

The initial weight states used in experiments with the 121 architecture with bias are:

initial weight state 1:

node[1][0] from node[0][0] ra=0.013871,p=-0.324274

node[1][0] from node[0][1] ra=0.247383,p=-0.375301

node[1][1] from node[0][0] ra=-0.191366,p=0.034532

node[1][1] from node[0][1] ra=0.041948,p=-0.110370

node[2][0] from node[1][0] ra=0.447630,p=-0.328272

node[2][0] from node[1][1] ra=0.202231,p=-0.273583

node[2][0] from node[1][2] ra=0.138615,p=-0.131947

initial weight state 2:

node[1][0] from node[0][0] ra=-0.005234,p=-0.375301

node[1][0] from node[0][1] ra=0.382839,p=0.146474

node[1][1] from node[0][0] ra=-0.416105,p=-0.110370

node[1][1] from node[0][1] ra=0.383572,p=0.280236

node[2][0] from node[1][0] ra=-0.222770,p=-0.131947

node[2][0] from node[1][1] ra=0.483459,p=0.035386

node[2][0] from node[1][2] ra=0.411481,p=-0.348079

initial weight state 3:

node[1][0] from node[0][0] ra=0.265679,p=0.146474  
node[1][0] from node[0][1] ra=0.173452,p=0.417203  
node[1][1] from node[0][0] ra=0.267144,p=0.280236  
node[1][1] from node[0][1] ra=0.259880,p=-0.098834  
node[2][0] from node[1][0] ra=0.322962,p=-0.348079  
node[2][0] from node[1][1] ra=0.125477,p=-0.185324  
node[2][0] from node[1][2] ra=0.303385,p=0.285424

initial weight state 4:

node[1][0] from node[0][0] ra=-0.153096,p=0.417203  
node[1][0] from node[0][1] ra=0.433271,p=0.174520  
node[1][1] from node[0][0] ra=0.019761,p=-0.098834  
node[1][1] from node[0][1] ra=0.379208,p=0.081896  
node[2][0] from node[1][0] ra=0.106769,p=0.285424  
node[2][0] from node[1][1] ra=0.431547,p=0.369930  
node[2][0] from node[1][2] ra=0.194617,p=-0.144368

initial weight state 5:

node[1][0] from node[0][0] ra=0.366543,p=0.174520  
node[1][0] from node[0][1] ra=0.207953,p=-0.036485  
node[1][1] from node[0][0] ra=0.258415,p=0.081896  
node[1][1] from node[0][1] ra=0.489593,p=-0.373562  
node[2][0] from node[1][0] ra=-0.110767,p=-0.144368  
node[2][0] from node[1][1] ra=-0.299768,p=0.326930  
node[2][0] from node[1][2] ra=0.106311,p=0.458464

## 2. Testing the training algorithm with a random I/O problem.

### Experiment 1

initial weight state	original solution	no. of epochs	error at the end of training
state 1	n/a	911,000	$<10^{-8}$
state 2	n/a	885,000	$<10^{-8}$
state 3	n/a	1,250,000	$<10^{-8}$
state 4	n/a	6,630,000	$<10^{-8}$
state 5	n/a	867,000	$<10^{-8}$

Table 11. Training a 121 architecture with four random patterns

The initial weight states used in experiments with the 121 architecture with bias are the same as above.

### Experiment 2

initial weight state	original solution	no. of epochs	error at the end of training
state 1	n/a	683,000	$<10^{-8}$
state 2	n/a	567,000	$<10^{-8}$
state 3	n/a	920,000	$<10^{-8}$
state 4	n/a	21,211,000	$<10^{-8}$
state 5	n/a	499,000	$<10^{-8}$

Table 12. Training a 131 architecture with four random patterns

## **Appendix 5**

### **Software documentation**

#### **1. The simulation software for the Complex Backpropagation**

##### **1.1. Introduction**

The software written for testing and experimenting with complex backpropagation was written in ANSI C and consists of a number of "C" source files, a number of *make* files (all the files associated with building an executable file under a specific operating system) and, lastly, some simulation files. Some information about each of these components will be given in the following.

In general, the software is completely portable and can run under any environment in which there are tools like a *make* program and a C compiler. The only parts of the software which depend on the environment are the makefile and eventually other command files like C-shell scripts, BAT files, etc. Of course, the use of such files is not compulsory and a version appropriate for the operating system in use can always be written using one of the set of files provided. The software has been tested under UNIX and DOS operating systems and command files for these environments are provided.

All the files forming the software can be put in a unique directory. However, it is recommended to create a directory structure which reflects the use of the software. Thus, if the software is used for several training simulations and/or problems, it is very convenient to create a sub-directory for each of these simulations. The files containing the network structure and the simulation information can have a standard name facilitating the use of the software. Alternatively, a specific extension can be associated with each type of file and the name of the file can be used to indicate the particular experiment a file belongs to. The existing files have been named according to the latter method as will be explained in the following.

##### **1.2. The source files**

###### **1.2.1 Where they are.**

If a directory structure is created, it is convenient for the source files to be kept in a base directory. The base directory is the directory which will contain all other



directories containing parts of this software and/or simulation environments. The base directory can be any directory in the operating system. The user must have read/write rights in this directory and in all sub-directories included for he/she to be able to use all the facilities of these programs.

### **1.2.2 The source file names and what they contain.**

The source files are:

**main.c** This contains the main function of the program. This file is also used to declare all the global variables so that they are accessible to functions in all source files.

**init.c** This contains all functions which initialise the software environment such as: initialisation of the weights, initialisation of the error structures, initialisation of the simulation environment, etc.

**cbp.c** This contains all functions which are used in the implementation of the complex backpropagation weight changing mechanism. If a different weight change mechanism is to be added or the current mechanism needs modifications, this is the only file which needs to be modified. All other functions are independent of the weight mechanism used (as long as the new mechanism does not need data which is not available in the system).

**struct.h** This contains the declarations of all variables, structures and other data types used in the programs.

**prototyp.h** This contains the declarations of all functions (prototypes) used in the program. This file is included in all the others and allows the compiler to check the types and the correct use of function parameters. Any modification of a function's parameters and/or return value should be updated in this file.

## **1.3 The *make* files.**

The generic name of 'make files' will be use to designate all the files necessary for building an executable version of the program from the source files (\*.c and \*.h). The *make* files may or may not involve using a *make* program.

### **1.3.1 Where they are.**

These files should be kept in the same directory in which the source files are. Eventually, the executable file could be kept in the same directory but this is not compulsory.

### 1.3.2 Their name and their use

Obviously, the name and the content of the *make* files depend on the operating system. This is because the *make* files contain commands which are specific to each operating system. In the following, two environments will be described: UNIX and DOS.

#### The UNIX operating system

Under this operating system, a unique *make* file exists and is used to describe the structure of the program and the commands to be executed when the executable version of the program becomes out of date due to modifications in at least one of the source files and/or the *make* file itself.

The file which contain these dependencies and commands is called "Makefile" and is read and interpret by the Unix program *make*. In order to build the program, the user must type:

```
>make
```

The *make* utility will check the dependencies in "Makefile" and will build the program if necessary.

#### The DOS operating system

Under this operating system, there exist two *make* files: CBP.MAK and CBP.LNK. The file CBP.MAK is the equivalent of the UNIX Makefile and contains all the dependencies between various files and the commands which must be executed in order to build an executable. These commands involve the C compiler available under DOS and the LINK utility. The versions actually used were Microsoft C Compiler versions 5.1 and Microsoft Link Utility version 3.1 but the software was tested without any problems with other compilers such as Microsoft Visual C++ version 1.0. The file CBP.LNK includes the command line for the LINK utility with all its options and input files.

In order to build the program, the user must type:

```
c:\>make cbp.mak
```

The *make* utility will check the dependencies and build the executable version of the program.

If a *make* utility is not available, the user could edit Makefile in UNIX or CBP.MAK in DOS and delete the lines describing the dependencies (the lines containing the symbol ":"). The result could be saved in a file called "BUILD" for instance. Subsequently, "build" will be the only command necessary for building the program. The only difference between using the *make* utility and such a 'build' file is that the 'build' version will always compile all files whereas the 'make' version will compile only the minimum number of files.

## **1.4. The simulation state files (SimState)**

### **1.4.1 Their use and what they contain.**

The state files are used to preserve the state of a simulation at a given moment. These files make it possible to stop the training at any time, save the state of the system in a simulation file and continue later, after restoring the state of the training session from the file. The files are ASCII files so they can be created or modified manually as well as from within the program.

A complete set of state files is constituted of a network file, a simulation file and a weight file. The network file contains information about the architecture of the network such as the number of layers and the number of neurons on each layer. The simulation file contains various training parameters and the I/O patterns. The weight file contains the weight state of the network. The state information was split into three files: architecture, patterns and weights because this offers more flexibility. Thus, the same architecture can be used with different pattern sets and from different initial weight states. Furthermore, the same set of patterns could be trained using different architectures and starting from different weight states.

However, it is necessary that the files are always loaded in the order: network file, simulation file and weight state file. The network file must be loaded before the simulation file because the loading of a simulation file needs a network structure which is created when a network file is loaded. At the same time, when a simulation file is loaded into a pre-existing architecture, the weights are initialised to random values. Therefore, loading a weight file before loading the desired simulation file is useless because the weight values will be lost. The responsibility of respecting this order is left entirely to the user.

### **Warning:**

Loading a simulation file without having loaded (or manually created) a network structure will crash the program and may crash the operating system. A crash of the operating system may cause data to be lost and inconvenience other users.

Loading a simulation file initialises the weights. Therefore, any weight state previously loaded or trained will be lost. If the current weight state is useful make sure it is saved before loading a simulation file.

Loading (or manually creating) a new network structure results in the loss of all information contained in any previously trained or loaded weight state or simulation. If such information is useful, make sure it is saved before loading (or creating) a network structure.

#### **1.4.2 Their structure.**

The structure of the state files will be presented in the following. Any manually created file must respect the same structure in order to be read and interpreted correctly by the program.

a) The network file contains the following elements in the following order:

1. The total number of neurons in the network.
2. The number of layers of neurons (including the first layer).
3. A number of integer values equal to the number of layers in the network. Each value represents the number of neurons on a layer of the network beginning with the first layer.

b) The simulation file contains the following elements in the following order:

1. The number of patterns
2. The number of input units
3. The number of output units
4. An integer which can be used as a seed for the random initialisation of the weights.
5. The error limit.
6. The coefficient controlling the shape of the sigmoid. For large values of this parameter, the sigmoid is close to a step function. For small values, the sigmoid will be close to a linear function.

7. The learning rate used to modify the radius values.
8. The learning rate used to modify the phase values.
9. The momentum parameter. The same value will be used for both radius and phase values.
10. The radial value of the bias.
11. The phase value of the bias.
12. A binary value (0 or 1) which specifies the type of input values. Zero corresponds to real and imaginary values and one corresponds to radius and phase values. This value must be 1 for the current version of the software.

c) The weight file contains the weight values. A weight has a radial value and a phase value, each of them as a double precision floating point number. The weights appear in the order of the neurons on layers starting with the first hidden neuron. The last weight for each unit is the weight coming from the bias unit.

Because a weight file is rather difficult to build manually from scratch, it is recommended that an existing file is modified instead. A 'dummy' file for the desired architecture is easily created by saving the initial (random) weight state immediately after loading the desired simulation file. This file can be modified manually and reloaded in the program.

## **1.5. Using the software**

### **1.5.1 Introduction**

This program is menu driven. In general the menu options are self-explanatory and the only responsibility which falls to the user is to respect the order of the files when loading them. Load first a network file, then a simulation file and then, if desired, a weight file.

### **1.5.2 Loading a network**

When this option is chosen, the program asks for the name of the file which contains the network structure. If the file is specified by typing just the file name as opposed to the complete path-name, the system will look for this file in the current directory first and in the directories specified by the environment variable "PATH" afterwards.

The file is expected to have the structure described in 1.4.2. The total number of neurons read from the file is checked against the sum of the neurons on each layer

as specified in the file. If the sum does not match the total number of neurons the error is reported and the program aborts.

### **1.5.3 Loading a simulation environment**

When this option is chosen, the program asks for the name of the file which contains the simulation information as described in 1.4.2.

The same network can be used to train many training sets. Consequently, the same network structure file can be used with many simulation files. However, the information contained in the simulation file is expected to be compatible with the network structure previously loaded. Such compatibility refers for instance to the number of input units in the network structure which should be equal with the number of input values in the patterns.

### **1.5.4 Training**

When this option is selected, a training session will be launched with the learning parameters specified in the simulation file.

The training uses some parameters which determine how often the training state is reported and saved. Thus, there are parameters such as the save step, the visualisation step and the stop step.

**The save step.** The state of the current training session is saved periodically to allow resuming the training after a system crash. The current weight state is saved in a file called "weights.saved" in Unix and "weights.sav" in Dos. If the operating system or the computer crashes during a simulation, the training can be resumed from the last saved weight state. A small value of the save step will instruct the program to save the weight state very often. A large value of this step will instruct the system to save the weight state less frequently. Saving the weight state frequently is not necessarily a good idea. There are at least two aspects. One of these is that the system will spend a lot of time saving its weight state instead of training. Secondly, if the ratio between the time used for effective training and the time used for saving the weight state is high, there is a higher probability for the system to crash during such a save in which case, the current training state cannot be recovered. Initially, the save step is given the default value 1000.

**The stop step.** The system can be required to stop after a given number of iterations for the user to inspect its state and possibly change the training parameters. The

value of the stop step specifies this number of iterations after which the system will stop requiring the user's intervention. Usually, this parameter should be given a large value (possibly the time-out limit).

The visualisation step. The system reports various error values from time to time. The visualisation step specifies the number of iterations after which the system will display the errors. These errors are the error for each pattern and the maximum error up to the current pattern. Both the radial and phase values are given. The display of the errors could take a lot of space if there are many patterns. This should be taken into consideration if the program is run in the background and the output is redirected into a file. At the same time, the error display slows down considerably the actual training of the network. A small value of this parameter is recommended only if the user is interested in following the evolution of the error on-line or a great amount of data is necessary for plotting the training curve very exactly.

#### **1.5.5 Saving the weight state**

When this option is chosen, the program asks for the name of the file in which the weight state is to be saved and saves the weight state of the network in a file with that name. If the file does not exist, it will be created. If the file exists, it will be overwritten.

#### **1.5.6 Loading a weight state**

When this option is chosen, the program asks the name of the file which contains the weight state information as described in 1.4.2. and loads it. If the file does not exist, the program reports the error.

#### **1.5.7 Testing**

The network can be presented with input patterns different from those used in the training (for testing generalisation for instance). These patterns must be in a file, one pattern per file. When the "test" option is chosen from the main menu of the program, the program asks for the name of the file from which such a test pattern is to be loaded. Then, the program loads this file and displays the output of the network yielded by the pattern read from the file.

## 1.6 A Tutorial for Complex Backpropagation

### 1.6.1 Introduction.

This tutorial presents an example session with the software simulator for neural networks.

The example session will take the reader through all the commands likely to appear in a normal work session with the simulator.

The commands the user has to introduce are printed in bold like **make cbp**. Every command has to be followed by a <CR> (or Carriage Return, or Return, depending on the keyboard). An example session can be run by typing in only the commands in bold. If the command is a filename or a character it appears as "**filename**" or "**1**". The quotes (") must NOT be typed in.

### 1.6.2 Using the simulator

#### 1. Getting in the right place.

Type:

```
tamdhu> cd /user.rsch/rsch/sorin/cbp/complex_bp
```

in order to change the current directory to the directory in which the simulator resides.

In this directory, there are two sub-directories called `complex_bp` and `standard_bp` respectively. The directory `complex_bp` contains the files used in performing the experiments with the complex backpropagation and their results. The directory `standard_bp` contains the files used in performing the experiments with the standard backpropagation and their results. The tutorial will use the files in the directory `standard_bp`. The use of the files in the directory `complex_bp` is very similar.

#### 2. Making the executable.

Type:

```
tamdhu> make cbp
```

The make program will check the dependencies and will build the program if necessary. Usually, the make program will give a message reflecting the fact that



the program is up to date. Note that write privileges are necessary for the program to be updated if the current version is out of date.

### **3. Running the simulator**

Type:

```
tamdhu> cd standard_bp
```

```
tamdhu> ls -l
```

A list of all the files in the directory will be displayed. The files with the extension ".net" contain architecture information, the files with the extension ".inp" contain simulation information and the files with the extension ".wei" or containing "weights" in their names contain weight information. Furthermore, there are other types of files: report files and subgoal files. These files are not relevant for the standard and complex backpropagation but have to be used. A subgoal file defines the subgoals for a Constraint Based Decomposition training when the training is directed by subgoal. For use with the complex backpropagation and the standard backpropagation, the subgoal file has to contain just one line with the number of patterns in the training set. This instructs the network to include all the patterns available in the first subgoal training session which in these cases is the only one. The report files contains in general information about the evolution of the error during a CBD training. The typical pieces of information are: the maximum error for the patterns in the current subgoal, the maximum error for the patterns in the previous subgoal, the number of patterns in the previous subgoal which generate an error higher than the error limit, the number of patterns in the current subgoal which generates such errors, etc. This information can be ignored for the standard and complex backpropagation if the CBD training is not used.

Launch the simulator with the command:

```
tamdhu>../cbp
```

The "." is necessary because the program is kept in the parent directory of the current directory.

The simulator displays its main menu:

- 1- load net
- 2- load simulation
- 3- learn
- 4- save weights
- 5- load weights
- 6- test
- 7- print weights
- 8- exit
- d- rbd

Type "1" for loading a net. The program will ask for the name of the file containing the architecture of the network. Type the name of the file you want to use. This file can be any of the files with the extension ".net" or your own file with the structure described in 1.4.2. For instance, you can type "**standard.net**" which contains the description of a network with 15 neurons organised on 3 layers in the following way: 2 inputs, 12 hidden units and 1 output unit.

After loading the file, the program will display the main menu again. Now, load a simulation file by typing "2". The program will ask for the name of the file containing the simulation information to be loaded. Type the name of the file you want to use. This file can be any of the files with the extension ".inp" or your own file with the structure described in 1.4.2. For instance, you can type "**standard.inp**" which contains the patterns presented in table 1. These values are 16 samples of the input signals presented in figs. 1 and 2 scaled in a suitable manner so that the resulting values of input 2 are between 0.5 and 1. The original samples and the scaling are given in chapter 8, section 8.5.1.1. The file "standard.inp" has zero values for the phase component of the input and the output patterns. This means that the program will effectively use the standard backpropagation.

training patterns		
input 1(time)	input 2	output
0	0.75	0.9
0.0625	0.805413	0.8334394
0.125	0.71464465	0.68535535
0.1875	0.51210155	0.5025994
0.25	0.3	0.35
0.3125	0.1909905	0.27881535
0.375	0.243934	0.30251265
0.4375	0.43018235	0.3931769
0.5	0.65	0.5
0.5625	0.78956175	0.57869265
0.625	0.78535535	0.61464465
0.6875	0.6586091	0.6266899
0.75	0.5	0.65
0.8125	0.41403475	0.7090526
0.875	0.456066	0.79748735
0.9375	0.599107	0.87753375

Table 1. I/O values for the standard network

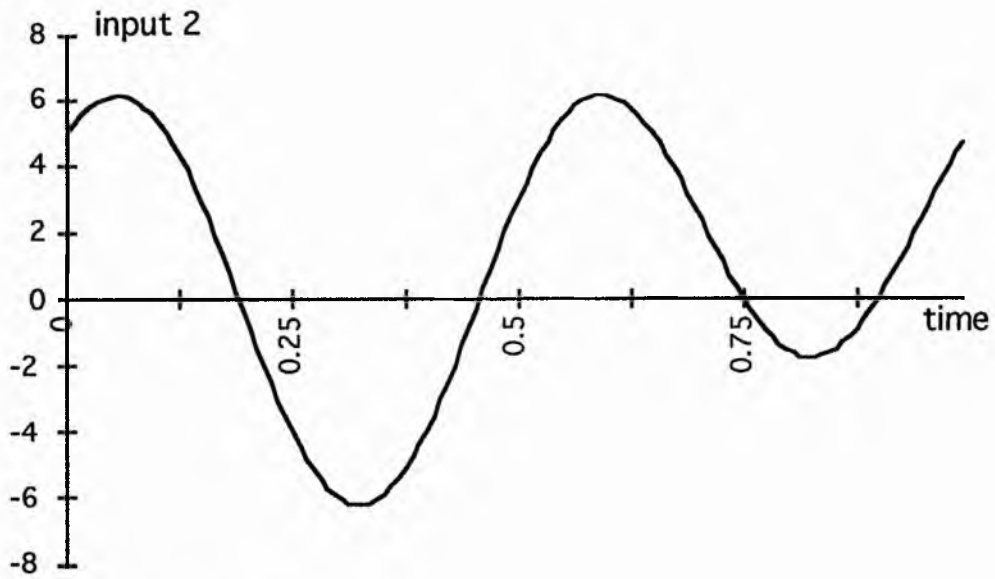


Fig. 1 The input signal

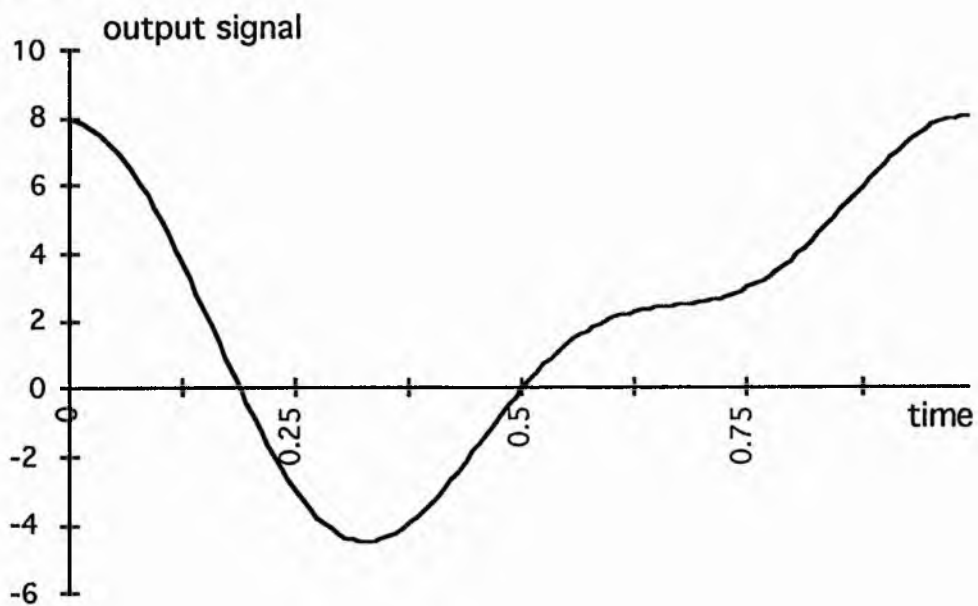


Fig. 2 The output signal.

Type "d" to launch a training session. The program will ask for the name of a subgoal file. As explained, this file should contain only a line with an integer value equal to the number of patterns in the training set. This tells the program that the training will be performed in just one subgoal session, whose training set contains

all patterns. Type **"standard\_subgoal"**. This file already exists and contains the integer value 16.

The program asks then for the name of a report file. As this information is not relevant for the standard and complex backpropagation training, type in any name of a file. You can type in for instance **"junk\_report"**. Be careful not to type in the name of an existing file which contains useful information. If you do this, the content of this file will be lost. By using the same name such as "junk\_report", you will not have to delete the file explicitly at the end of the training session because each training session will overwrite the file created in the previous session.

The program will ask for the name of the file in which the weights will be saved at the end of each subgoal session. This will be the name of the file in which you will find the trained weights at the end of the training session if the training session is successful. Type in any file name, for instance **"my\_weights.wei"**.

After this name is typed in, the program launches the training. Almost immediately, you will see something like:

```
cycle=0,error=0.364353,err=0.364353,pattern=0
cycle=0,error=0.364353,err=0.298087,pattern=1
cycle=0,error=0.364353,err=0.149607,pattern=2
cycle=0,error=0.364353,err=0.034075,pattern=3
cycle=0,error=0.364353,err=0.187647,pattern=4
cycle=0,error=0.364353,err=0.259315,pattern=5
cycle=0,error=0.364353,err=0.235334,pattern=6
cycle=0,error=0.364353,err=0.143754,pattern=7
cycle=0,error=0.364353,err=0.035857,pattern=8
cycle=0,error=0.364353,err=0.043529,pattern=9
cycle=0,error=0.364353,err=0.079494,pattern=10
cycle=0,error=0.364353,err=0.090973,pattern=11
cycle=0,error=0.364353,err=0.113564,pattern=12
cycle=0,error=0.364353,err=0.172243,pattern=13
cycle=0,error=0.364353,err=0.260910,pattern=14
cycle=0,error=0.364353,err=0.341666,pattern=15
```

These are the errors for each pattern (err) together with the maximum error up to the current pattern (error). In the "standard.inp" file there are 16 patterns numbered

from 0 to 15. After this "report per cycle", the program stops the training and displays another menu (the training menu):

s-stop step

v-visualising step

r-radius learning rate

b-beta

m-momentum

p-phase learning rate

q-quit

If you press carriage return, the program will perform another cycle and display this menu again. This 'controlled' training is useful when the user wants to follow a critical phase in the training. However, in general it is better if the program executes more than one cycle before stopping. The number of training cycles (or epochs) performed by the program before it stops and waits for input is controlled by the "stop cycle". This stop step can be set with the option "s". The default value of the stop step is 1.

Analogous to the stop step, there is a parameter called "visualising step" which controls the number of epochs after which the program displays a "report per cycle". The default value of the visualising step is 1.

Type "v" to change the visualising step. The program will prompt for the new value of the visualising step. Note that a small value of this parameter makes the program display a lot of cycle reports which can scroll too rapidly on your screen and/or occupy a lot of space on the disk (if you redirect the output or you enable the scroll bar of the command tool you are using). Use a large value unless you are really interested in following closely the evolution of the error. Type in "1000" for instance.

The program will then display again the training menu:

s-stop step  
v-visualising step  
r-radius learning rate  
b-beta  
m-momentum  
p-phase learning rate  
q-quit

No errors should be displayed this time. This is because the stop step has still the default value "1" whereas the visualising step has the value 1000. This means that the program will stop after each epoch but will display the errors only every 1000 epochs.

Type "s". The program prompts for a value. Type in a very large value, for instance "10000000". This means that the program will stop after 10 million epochs or when the error goes below the error limit, whichever comes first.

Note that the visualising step was changed before the stop step. Let us try to imagine what happens if the stop step is changed before the visualising step. Initially, both of them have the default value 1. This means that the program displays the errors after each epoch after which it stops waiting for input. Let us assume that the stop step is changed first to the value 10,000,000. The program will display the errors at the end of each epoch but will not stop until the error goes below the error limit or 10,000,000 epochs are executed. In practice, you will see the screen filling with error reports and starting to scroll. This will continue for some time which can be very long. If you get in such a situation and you want to interrupt the training you can press "^C" (CONTROL-C). At this stage (i.e. during the training), "^C" does not abort the program but it interrupts the training instead. The program will display the training menu again, giving you a change to modify the visualising step or any other training parameter.

You can use the "^C" command at any time during the training if you want to change any of the training parameters like learning rate, momentum or beta (the parameter adjusting the shape of the sigmoid, see also chapter 2 equation 13). For the standard backpropagation, the radius learning rate plays the role of the learning rate and the phase learning rate does not have any effect.

If at any stage you want to save the state of the network (in order to resume later or to use it as a backup) you should type "**^C**". You don't have to type "**^C**" if the program is waiting for user input. In such a case, just choose the appropriate option(s). Let us assume the training was running and you pressed "**^C**". The program will display the training menu:

- s-stop step
- v-visualising step
- r-radius learning rate
- b-beta
- m-momentum
- p-phase learning rate
- q-quit

Type "**q**". The program will go up a menu and will display the main menu:

- 1- load net
- 2- load simulation
- 3- learn
- 4- save weights
- 5- load weights
- 6- test
- 7- print weights
- 8- exit
- d- rbd

Type "**4**". The program will prompt for a filename. Type in the name of the file you want the weights to be saved, for instance "**my\_weights**". The difference between "**my\_weights**" and "**my\_weights.wei**" (the name of the file typed in when the training session was started) is that "**my\_weights**" will contain the weights at the moment the user chooses option 4 (save weights) whereas "**my\_weights.wei**" will contain the weights at the end of the training only if the training was successful.

At this stage, the training can be interrupted by typing "**8**". In this phase, the training is just suspended and it can be continued. You could for instance, have a look at the weights (by printing them), save them in a file and go back to training. This is useful if you are not sure that the training will go the right way. Remember that the weights are always saved in the same file "**weights.saved**". If you get an



acceptable error at some stage during the training, you might want to save the weights in a different file just in case the training will diverge later.

If you want to resume the training after you quit the program, run the program again, load the same network architecture file and simulation file and choose option 5 (load weights). Then, give the name of the file the weights have been saved in.

Option 7 (print weights) displays the weights on the screen. This is the only advisable way to inspect the weights. The weight file (\*.wei) was not designed to be used directly by the user. The user is supposed to consult the weights from the program. If you want to see the content of the weight file, you are supposed to load it in the program first.

The phase weights are present for uniformity. They are just ignored during the processing. Note that they remain at the values they are initialised.

The layout used for displaying the weights is displayed in the following. The comments are marked by a string of "\*\*\*". The rest is what the program actually displays. Three dots (...) indicates that lines were omitted.

First, the structure of the network is given:

```
layers_num=3
total_num=15
layer 0 has 2 neurons
layer 1 has 12 neurons
layer 2 has 1 neurons
```

\*\*\*\*\* from here...

```
node[0][0] to node[1][0] ra=0.121223,p=-0.324274ra=0.121223,p=-0.324274
```

....

```
node[0][0] to node[1][11] ra=0.351879,p=-0.185324ra=0.351879,p=-0.185324
```

\*\*\*\*\* ...to here, the program displayed the weights from unit 0 on layer 0 to all units on the next layer (as reported above, there are 12 units on the second layer numbered from 0 to 11)

\*\*\*\*\* from here...

node[0][1] to node[1][0] ra=0.113738,p=0.417203ra=0.113738,p=0.417203

node[0][1] to node[1][1] ra=0.555872,p=-0.098834ra=0.555872,p=-0.098834

...

node[0][1] to node[1][10] ra=-0.402724,p=0.458464ra=-0.402724,p=0.458464

node[0][1] to node[1][11] ra=0.831917,p=-0.090960ra=0.831917,p=-0.090960

\*\*\*\*\* ... to here, the program displayed the weights from unit 1 on layer 0 to all units on the next layer

\*\*\*\*\* from here

node[1][0] to node[2][0] ra=0.218501,p=0.257897ra=0.218501,p=0.257897

\*\*\*\*\* to here, the program displayed the weights from unit 0 on layer 1 to all units on the next layer (as reported above there is only 1 unit on the output layer)

\*\*\*\*\* from here

node[1][1] to node[2][0] ra=0.628643,p=-0.471923ra=0.628643,p=-0.471923

\*\*\*\*\* ... to here, the program displayed the weights from unit 1 on layer 1 to all units on the next layer (as reported above there is only 1 unit on the output layer)

\*\*\*\*\* from here

node[1][2] to node[2][0] ra=-0.017186,p=0.256951ra=-0.017186,p=0.256951

... to here, the program displayed the weights from unit 2 on layer 1 to all units on the next layer (as reported above there is only 1 unit on the output layer)

node[1][3] to node[2][0] ra=0.146677,p=0.089557ra=0.146677,p=0.089557

node[1][4] to node[2][0] ra=-0.382791,p=0.456053ra=-0.382791,p=0.456053

...

node[1][9] to node[2][0] ra=0.628880,p=-0.263268ra=0.628880,p=-0.263268

>>node[1][10] to node[2][0] ra=-0.311320,p=-0.093921ra=-

0.311320,p=0.093921

>>node[1][11] to node[2][0] ra=0.874825,p=-0.073046ra=0.874825,p=-

0.073046

\*\*\*\*\* the weights from the second to the third layer are finished

\*\*\*\*\* The bias weights follow. The neurons are numbered in layer's order: the first 12 units are those on the second layer and the last unit is the one on the last layer. The input units do not have bias weights.

\*\*\*\*\*

bias weight to neuron[1][0] ra = -0.146486 p = -0.118030

bias weight to neuron[1][1] ra = -0.464171 p = -0.339412

...

bias weight to neuron[1][11] ra = 0.288638 p = 0.467223

bias weight to neuron[2][0] ra = 0.111606 p = -0.068499

Option 6 (test) instructs the program to prompt for a filename. This file is expected to contain just the input values of a test pattern. The program will read these values, calculate the output of the network using the current weight state and display it.

### **1.6.3 Running the program in the background**

In order to run the program in the background, all its input, output and error messages have to be redirected. The user must prepare a file containing all the commands the program might expect for instance. The possible content of such a file (input) is given below.

```
1
cbp.net
2
cbp.inp
d
subgoal
junk_report
final_weights
v
1000
s
3000
r
0.001
s
5000
p
0.01
q
4
weights_after_18000_epochs
8
```

Such a file will have the effect described in table 2. Now, the program can be launched in the background using:

```
tamdhu> cbp <input >output 2>error &
```

The output of the program will be put in the file "output".

Line in the redirected input file	Interpretation
1	load net file option
cbp.net	name of the file containing the net architecture
2	load simulation option
cbp.inp	name of the file containing the patterns
d	start training
subgoal	name of the subgoal file
junk_report	name of the report file (not important)
final_weights	name of the file in which the final weights will be saved
v	set the visualisation step...
1000	... to 1000 epochs
s	set the stop step...
3000	to 3000 epochs
r	change the radius learning rate...
0.001	...to 0.001 (this happens after 3000 epochs)
s	change the stop step...
5000	...to 5000 epochs (this happens after 3000+3000 epochs)
p	change the phase learning rate...
0.01	...to 0.01 (this happens after 3000+3000+5000 epochs)
q	quit training after 3000+3000+5000+5000 epochs
4	save weights as trained...
weights_after_16000_epochs	... in this file
8	exit the program

Table 2. The effect of the commands in the file described.

#### 1.6.4 More about the existing files

The directory `/user.rsch/rsch/sorin/cbp/complex_bp` contains the files used in performing the experiments described in chapter 8 and their results.

The sub-directory `standard_bp` contains files like:

`standard.net` = the architecture 2-12-1 used to train one and two signals as associations of point values.

`standard.inp` = 16 samples of one input signal associated with 16 samples of one output signal.

`2signals.inp` = 16 samples of two input signals associated with 16 signals of two output signals.

`standard8.inp` = 8 samples of one input signal associated with 8 samples of one output signal.

There are also files containing the weights at the end of the training for each of the training sets above.

The sub-directory `complex_bp` contains files like:

`cbp.net` = the architecture 6-6 used to train one and two signals as associations of parametric representations.

`cbp1point.inp` = the training patterns corresponding to the association of one input signal to one output signal.

`cbp2points.inp` = the training patterns corresponding to the association of two input signals to two output signals.

There are also files containing the weights at the end of the training for each of the training sets above.

In all experiments described in Chapter 8 which were performed with more than one weight state, the first one which is reported in the tables is the one which uses the weight state automatically initialised when the program is run for the first time. For instance, running the program with the net file "`standard.net`" and the simulation file "`standard.inp`" should lead to the first row in table 8 after the number of epochs reported.

## **2. Constraint Based Decomposition**

### **2.1. Introduction**

The software written for testing and experimenting the Constraint Based Decomposition was written in the "C" programming language and consists of a number of "C" source files and a 'make' file. Some information about each of these components will be given in the following.

This software implements Constraint Based Decomposition with or without various enhancements for problems with 2 inputs and one output. The dimensionality of the problem has been restricted to these values for the convenience of being able to represent the I/O mapping graphically in a plane. The I/O mapping of the trained network can then be displayed on the screen and/or printed.

There are two versions of this software: one for the IBM-PC compatible/DOS environment and one for the SUN/Unix/XView environment. There are small differences between the features offered by the two versions which will be explained in the following.

### **2.2. The source files**

#### **2.2.1 Where they are.**

If a directory structure is created, it is convenient for the source files to be kept in the base directory. The base directory is the directory which will contain all other directories containing parts of this software and/or simulation environments. The base directory can be any directory in the operating system. The user must have read/write rights in this directory and in all directories included for he/she to be able to use all facilities of this programs.

A complete set of source files can be found in the directory /user/rsch/sorin/macfiles/rbd/stat\_rbd

#### **2.2.2 The source files names and what they contain.**

The source files are:

**main.c**        This contains the main functions of the program. This is also used to declare all the global variables so that they are accessible to functions in all source files.

**x\_draw.c** This contains the functions specific to the XView version of the program. These functions are not used in the PC version.

**tools.c** This contains various functions used in drawing on the screen for the PC version

**solution.c** This contains functions which manage the solution abstract data type (building a solution, displaying a solution on the screen, saving it in a file, etc).

**hp.c** This contains functions which manage the hyperplane abstract data type.

**subgoal.c** This contains the functions which manage the subgoal abstract data type.

**rbd.h** This contains the declarations of all variables, structures and other data types used in the programs. Furthermore, this file contains a series of constants which control the version of the program which will be built.

**protrbd.h** This contains the declarations of all functions (prototypes) used in the program. This file is included in all the others and allows the compiler to check the types and the correct use of function parameters.

**Makefile** This contains a list of dependencies and the commands to be executed for building the target version of the program.

### **2.2.3 Building different versions of the program.**

The user can select the version of the program to be built by editing the file **rbd.h**. There are two main versions (PC/Sun) and several variations for each version. In the following there will be described various options which influence the version of the program to be built.

**TEXT\_IN\_RESULT** - If this constant is defined, the program will insert explanatory text in the output file containing the solution of the problem. This option is available on both PC and Sun versions.

**SHOW\_LOCKING** - If this constant is defined, the program will display the locking situations as they arise. This option is available only in the PC version.

**PC** - If this constant is defined, the PC version of the program will be built.



**SUN** - If this constant is defined, the SUN version of the program will be built. Only one of the two versions can be built at any one time. Therefore, only one of the constants PC and SUN can be defined. If both are defined, the compilation will fail with a series of "multiply defined symbol" linkage errors.

**REDUNDANCY** - If this constant is defined, the program will use the redundancy detection mechanism. This option is available in both PC and SUN versions.

**LOCKING** - If this constant is defined, the program will use the locking detection mechanism. This option is available in both PC and SUN versions.

All constants discussed above are already defined in the file rbd.h. Some of them are commented out. You can control the structure of the program being built by commenting out the constants you don't want and leaving in only those you want. Although the particular values of the constants are not important, it is advisable not to delete any definition. Commenting definitions out as opposed to deleting them has the advantage that next time the program has to be reconfigured, all the choices available will be already there.

The file rbd.h contains the definitions of other constants as well such as the maximum number of patterns a training set can contain, etc. These values are important. Therefore, it is recommended that the constants not discussed above remained unchanged. This is because, by changing definitions of various constants, the user modifies the amount of memory the program will need to run and in turn, this could cause "out-of-memory" run-time errors especially for the PC/DOS version of the program which has to fit in 640K.

## **2.3. Making the executable**

In order to make the executable, change the current directory to the directory which contains the source files (currently /user/rsch/sorin/macfiles/rbd/stat\_rbd) and type "make x\_draw". The "make" program will check the dependencies and will build the executable version if the current one is out of date. Usually this command will return the message: "x\_draw is up to date".

## **2.4. The model file**

The model file is a file containing the patterns to be trained. Every time the program is run, it will ask for the name of such a model file. A model file should contain the number of the patterns on the first line followed by the I/O values corresponding to

the patterns in the order: input 1, input 2, output. As the program accepts only patterns with 2 input values and 1 output value, this information need not be contained in the model file. The model file can contain more patterns than the number of patterns indicated on the first line of the file. In this case, the last patterns will be ignored.

## 2.5. Running the simulator

As the algorithm is not characterised by user-modifiable parameters, running the program is fairly simple and does not need the user's intervention. When launched, the program prompts for the name of the file containing the patterns to be trained. Once this file is read, the program does everything else automatically.

Once the solution has been built, the program displays it one term<sup>2</sup> at the time. When all terms of both classes have been displayed, the program builds and displays the I/O mapping corresponding to the solution. In general, every time the program displays some information it will wait for the user to acknowledge by pressing <return>.

## 2.6 A Tutorial for using the CBD simulation software

### 2.6.1 Introduction.

This tutorial presents an example session with the software simulator for neural networks which implements Constraint Based Decomposition.

The example session will take the reader through all the commands likely to appear in a normal work session with the simulator.

The commands the user has to introduce are printed in bold like **make x\_draw**. Every command has to be followed by a <CR> (or Carriage Return, or Return, depending on the keyboard). An example session can be run by typing in only the commands in bold. If the command is a filename or a character it appears as "**filename**" or "**1**". The quotes (") must NOT be typed in.

---

<sup>2</sup>See the chapter "The Constraint Based Decomposition" for an explanation of notions like "hyperplane", "factor", "term", "solution".

## 2.6.2 Using the simulator

### 1. Getting in the right place.

Type:

```
tamdhu> cd /user.rsch/rsch/sorin/macfiles/rbd/stat_rbd
```

in order to change the current directory to the directory in which the simulator resides.

This directory contains the source code, the executable version of the program as well as data files and result files.

### 2. Making the executable.

Type:

```
tamdhu> make x_draw
```

The make program will check the dependencies and will build the program if necessary. Usually, the make program will give a message reflecting the fact that the program is up to date. Note that write privileges are necessary for the program to be updated if the current version is out of date.

### 3. Running the simulator

Launch the simulator with the command:

```
tamdhu> x_draw
```

The program will prompt for the name of a file containing the patterns to be trained. Type "**goal.dat**" for instance. This file contains 20 patterns of the 2-grid problem discussed in Chapter 5. The program will start building the network and training its connections. The program gives some information regarding the training. For instance, the number of patterns in each subgoal, the number epochs necessary to train each subgoal and the success or the failure of each subgoal training are reported. Furthermore, the programs always gives information regarding the number of the hyperplane which is being trained.

When the training is finished, the program displays the solution.

>class 1:

This message shows that a description of the units which will fire for class 1 follows (the units on the AND layer). When all the units which fire for class 1 have been described, a similar message ("class 2:") will mark the start of the units for the second class.

As the implementation is limited to only two classes, there are no other messages of this kind.

An AND unit is described in the following way:

hidden unit 0  $w_x = -5.391023$ ,  $w_y = 2.065994$ ,  $w_b = 0.331628$  with +

The implementation is limited to 2 input units: x and y. For each unit on the AND layer the weight  $w_x$  is the weight coming from the x input unit,  $w_y$  is the weight coming from the y input unit and  $w_b$  is the weight coming from the bias unit. The sign at the end ("with +") is the sign of this unit on the AND layer and its meaning is described in Chapter 4 (The Constraint Based Decomposition).

In the same chapter, the solution is presented in the form of an expression like:

$$\text{class1} = h_1 * h_2 * h_3 + \sim h_1 * h_2 * h_4 + \sim h_1 * \sim h_3$$

where  $\sim$  means "not" (and was indicated by a horizontal bar on top of the hyperplane in Chapter 4). In the same expression "\*" stands for logical AND and "+" for logical OR. An expression like the above will be described by the program in the following way:

class1:  
hidden unit 1 <weights of h1> with +  
AND  
hidden unit 1 <weights of h2> with +  
AND  
hidden unit 1 <weights of h3> with +  
OR

hidden unit 1 <weights of h1> with -  
AND  
hidden unit 1 <weights of h2> with +  
AND  
hidden unit 1 <weights of h4> with +  
OR

hidden unit 1 <weights of h1> with -  
AND  
hidden unit 1 <weights of h3> with -

class2:  
... something similar

Unfortunately, in the actual output of the program there is a redundant "AND" just before the "OR". This is because the "AND" is printed automatically at the end of the description of a neuron. At this stage, there is no way to decide whether the description of a class is finished and an "OR" is needed or it will continue with another neuron in which case an "AND" is needed.

When the training is finished, the program plots the I/O mapping in a new window. Usually, this phase -which is signalled by the message "in repaint" takes some time. When this is done, the window is displayed on the screen.

If you desire to print the result of the training, use the "snapshot" application to write the result window in a file and print that file. Unfortunately, the colours displayed on the screen will be different from the colours printed on paper. Therefore, colour corrections may be needed and can be performed with an application like "xv".

### 2.6.3 More about existing files.

The directory `/user.rsch/rsch/sorin/macfiles/rbd/stat_rbd` contains the files used in performing the experiments described in chapter 5 and their results.

The following files can be found in this directory:

`goal.dat` = This file contains 20 patterns from the 2-grid problem described in chapter 5. The patterns are given in an arbitrary order.

`goal20_ordered.dat` = This file contains 20 patterns from the 2-grid problem described in chapter 5. The patterns are given in an order which will determine the network to find a minimal solution (using the minimum number of hyperplanes)

`goal20_ordered1.dat` - `goal20_ordered4.dat` = These files contain the same 20 patterns but the order is different in each of them. However, all of them determine the network to find a minimal solution.

`spiral.dat` = This file contains 194 patterns from the 2-spiral problem.

`dense.dat` = This file contains 770 patterns from the 2-spiral problem.

There are also some raster files containing the I/O mapping of the solution for some problems. These raster files have the extension ".rs" and they can be inspected using a program such as "xv" or any other utility able to read Sun raster file format.

Some raster files are:

`dense58.rs` = contains a solution (with 58 hyperplanes) of the 2-spiral problem trained with 770 patterns.

`dense42.rs` = contains a solution (with 42 hyperplanes) of the 2-spiral problem trained with 770 patterns.

`goal20.rs` = contains a solution of the 2-grid problem trained with an arbitrary order

`goal_ordered20.rs` = contains a solution of the 2-grid problem trained with an order designed to yield a minimal solution (with the minimum number of hyperplanes).

`32hp.rs` = contains a solution (with 32 hyperplanes) of the 2-spiral problem trained with 194 patterns.

## Appendix 6

### Some solutions found by the Constraint Based Decomposition Algorithm

#### General remarks

These images contain a number of patterns of two colours. The task of the algorithm is to build a neural network with two output units. Each output unit is associated with a class of the input patterns and must be active (output of 1) when patterns from its class are presented. The pictures represent the I/O mapping of the network constructed by the algorithm. in the square  $-10 < x < 10$ ,  $-10 < y < 10$ . The hyperplanes implemented by the units on the first hidden layer (the AND layer) are represented by a double line (white and black). The white side of the line indicates the half-hyperplane for which the output of the unit is positive.

**Picture 1.** A solution for the 2-grid problem obtained by using the perceptron training algorithm as the weight changing mechanism. Note that the dividing hyperplanes are placed very close to some patterns because the subgoal training stops when the patterns are correctly classified.

**Picture 2.** A different solution for the 2-grid problem obtained by using the delta rule as the weight changing mechanism. Note that the dividing hyperplanes are placed as far as possible from the patterns because the subgoal training attempts to reach analogue the patterns' analogue targets.

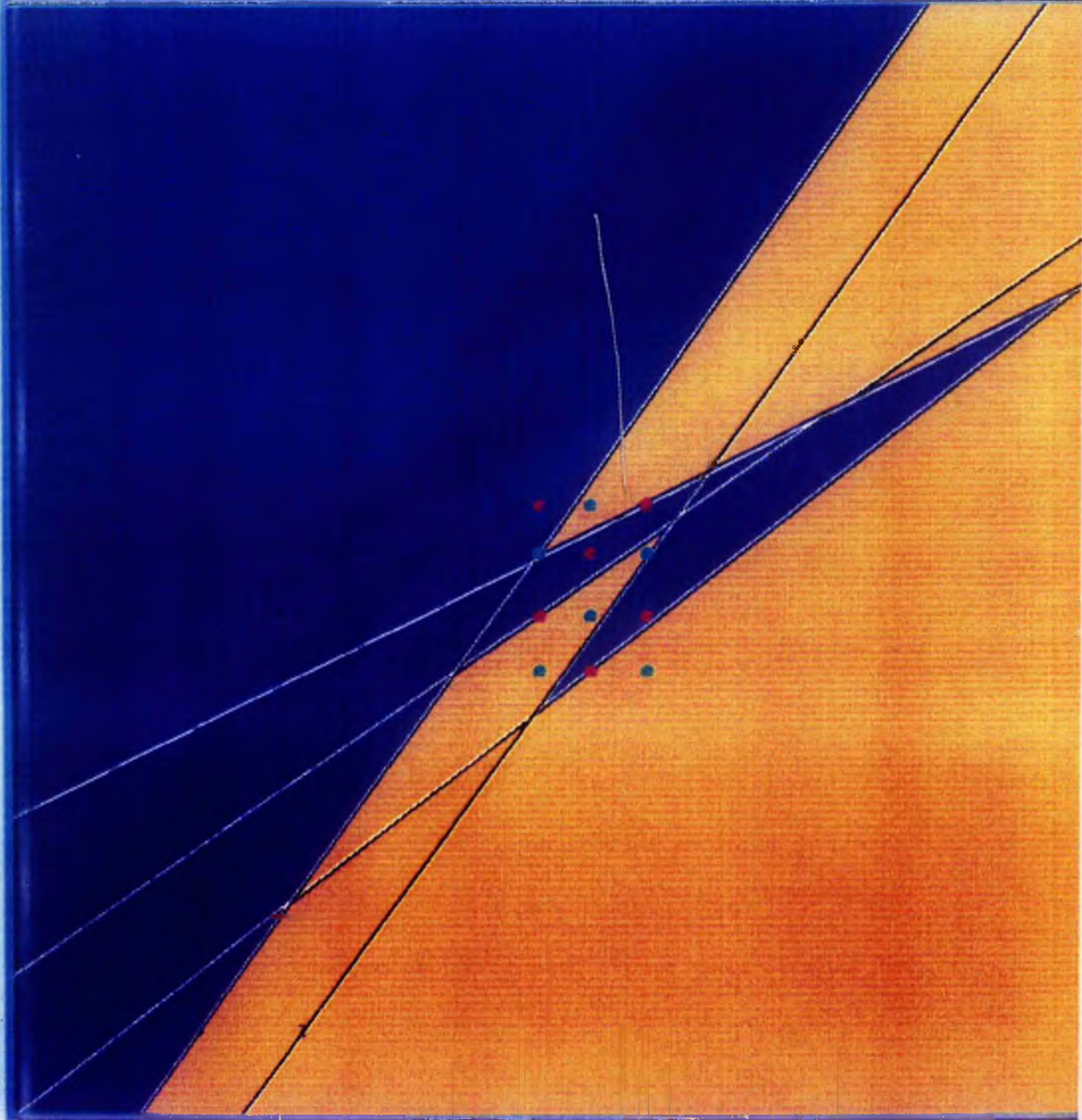
**Picture 3.** A solution for the 2-spiral problem (196 patterns) obtained by the simple version of the algorithm (no redundancy elimination) and the delta rule as the weight changing mechanism. Note the presence of redundant hyperplanes.

**Picture 4.** A solution for the 2-spiral problem (196 patterns) obtained by the enhanced version of the algorithm (with redundancy elimination) and the delta rule as the weight changing mechanism. Note the fact that the redundant hyperplanes have been eliminated.

**Picture 5.** A solution for the 2-spiral problem (784 patterns). The fact that the convergence of the algorithm is guaranteed allows the use of a sufficient number of patterns for the desired generalisation. Note that the 2 spirals are very clear.



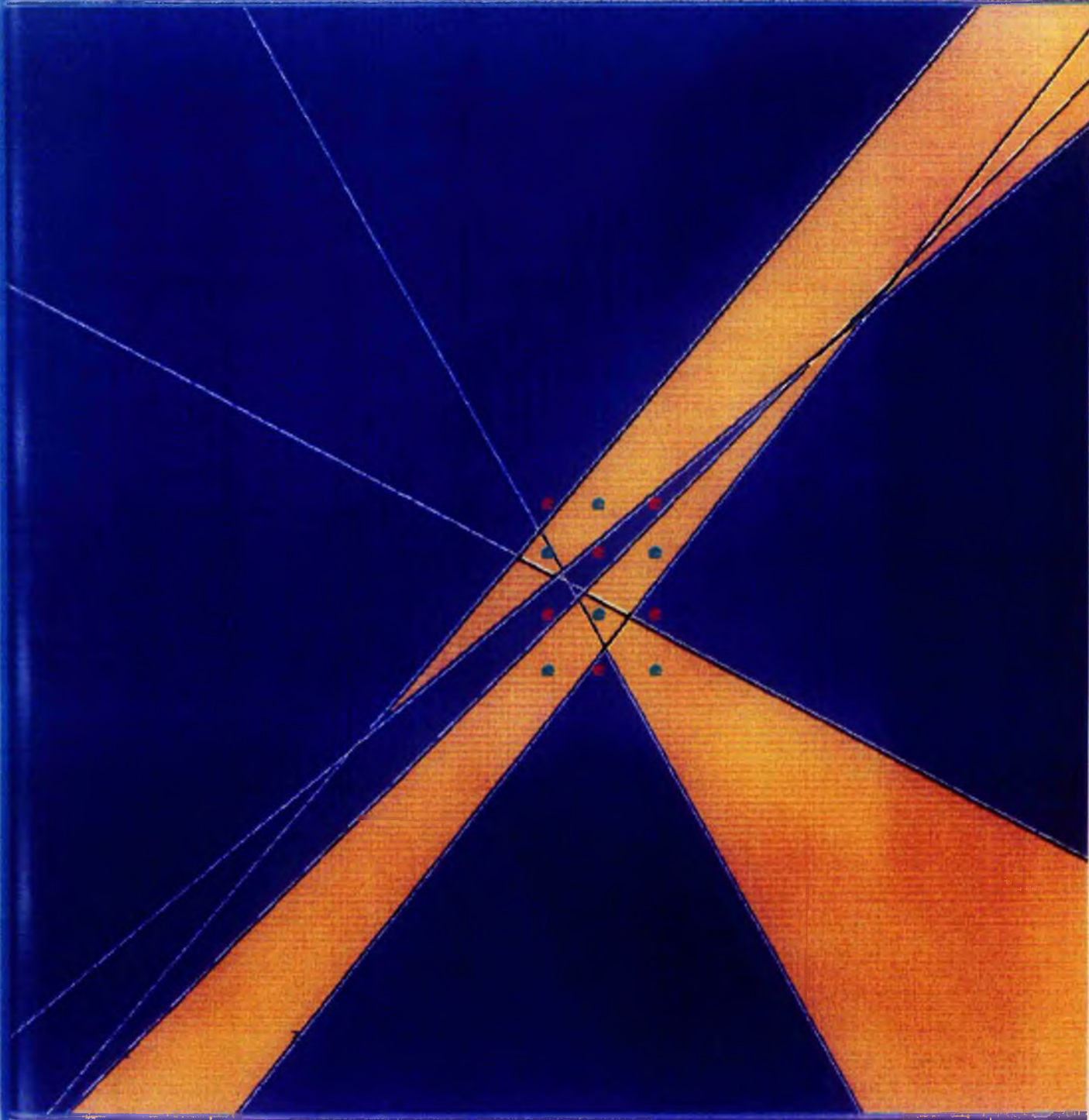
xv\_canvas\_x\_draw



A solution for the 2-grid problem obtained by using the perceptron training algorithm as the weight changing mechanism. Note that the dividing hyperplanes are placed very close to some patterns because the subgoal training stops when the patterns are correctly classified.

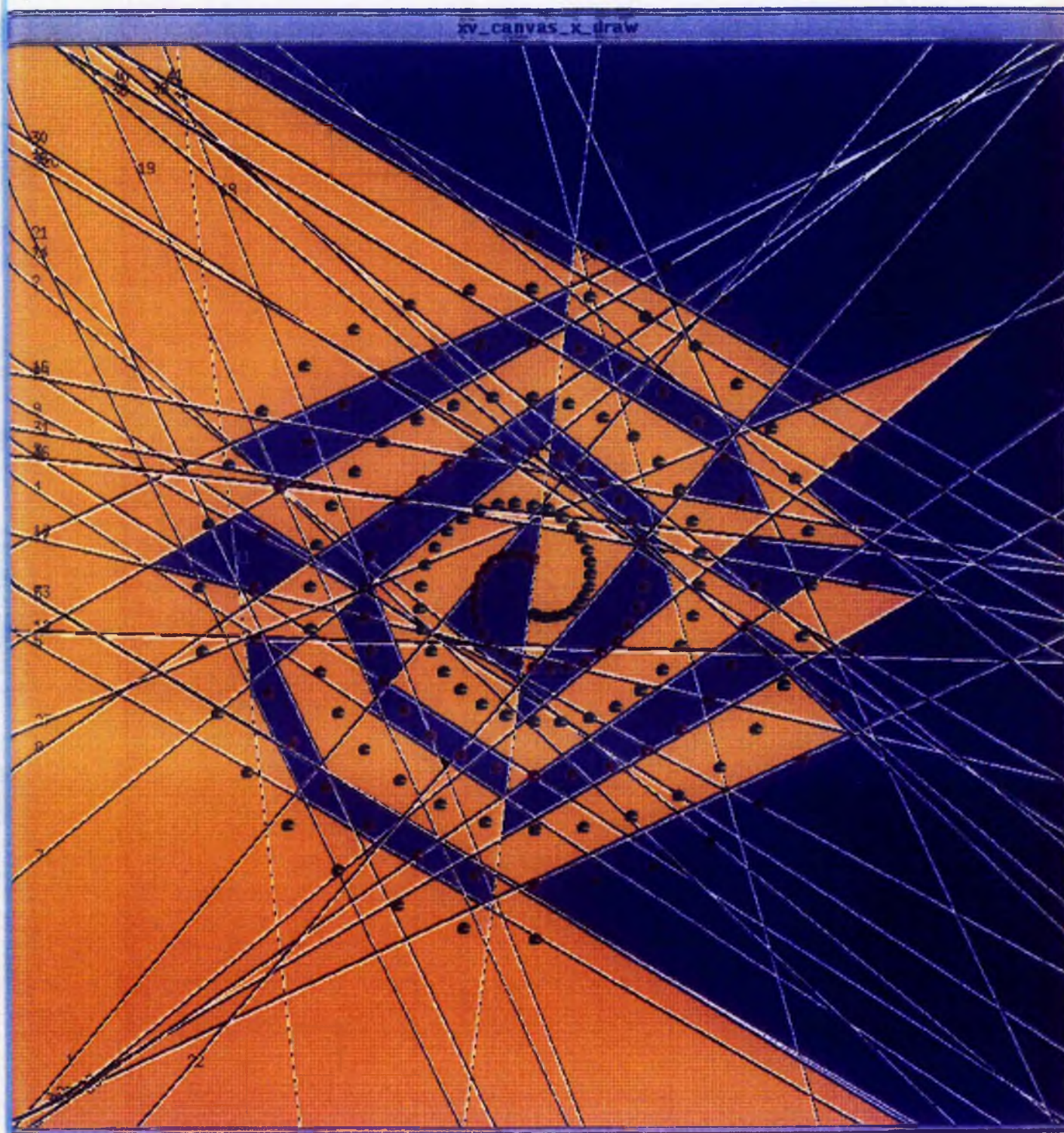


xv\_canvas\_x\_draw



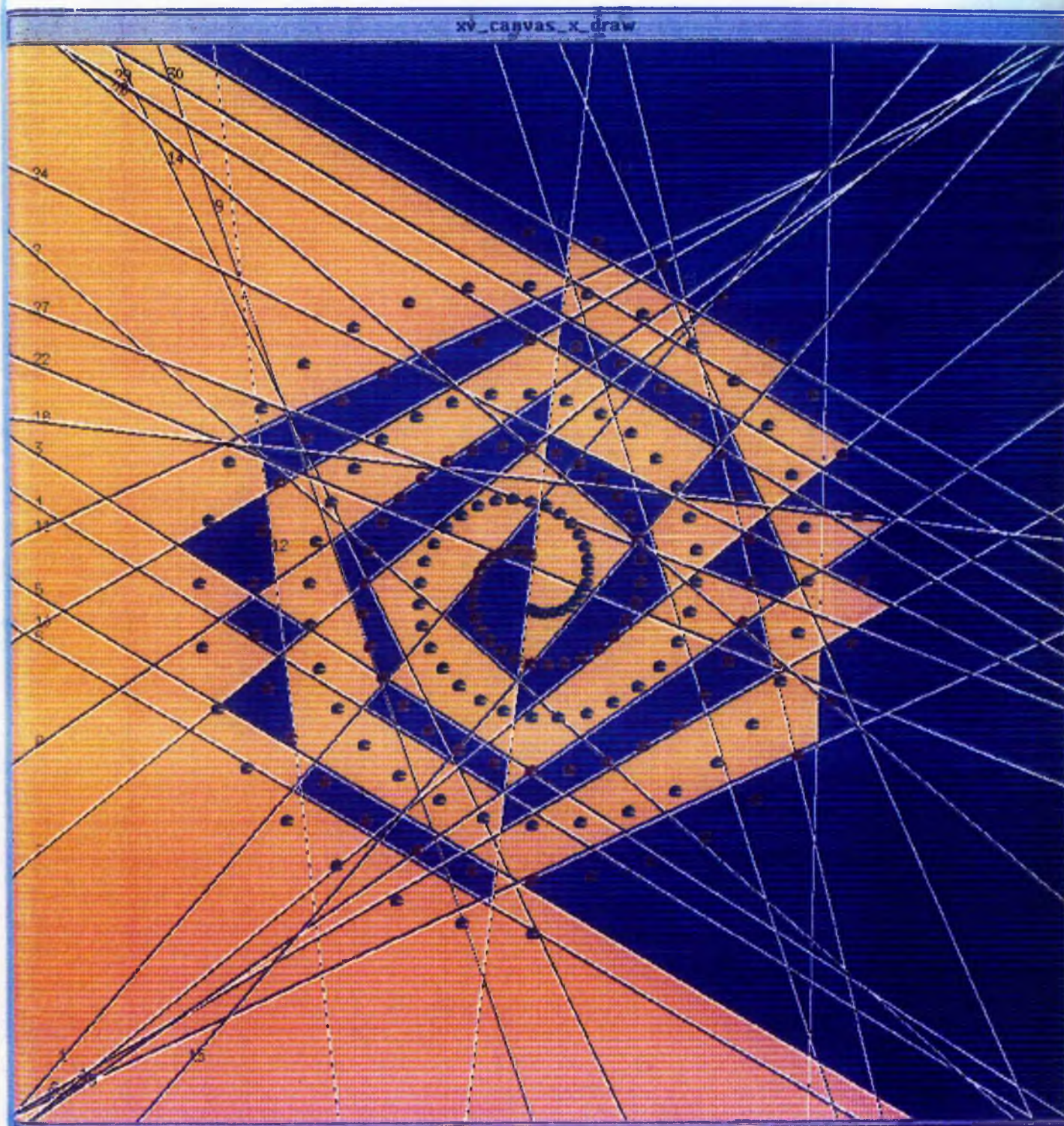
A different solution for the 2-grid problem obtained by using the delta rule as the weight changing mechanism. Note that the dividing hyperplanes are placed as far as possible from the patterns because the subgoal training attempts to reach ~~analogue~~ the patterns' analogue targets.





A solution for the 2-spiral problem (196 patterns) obtained by the simple version of the algorithm (no redundancy elimination) and the delta rule as the weight changing mechanism. Note the presence of redundant hyperplanes.





A solution for the 2-spiral problem (196 patterns) obtained by the enhanced version of the algorithm (with redundancy elimination) and the delta rule as the weight changing mechanism. Note the fact that the redundant hyperplanes have been eliminated.





A solution for the 2-spiral problem (784 patterns). The fact that the convergence of the algorithm is guaranteed allows the use of a sufficient number of patterns for the desired generalisation. Note that the 2 spirals are very clear.